Федеральное государственное бюджетное образовательное учреждение высшего образования

Национально исследовательский университет «Московский энергетический институт»

Кафедра ВМСС

Лабораторная работа №3

по курсу

«Программное обеспечение высокопроизводительных вычислительных систем»

Выполнили: студенты группы А-07м-23

Балашов С.А., Кретов Н.В., Михайлов С.А.

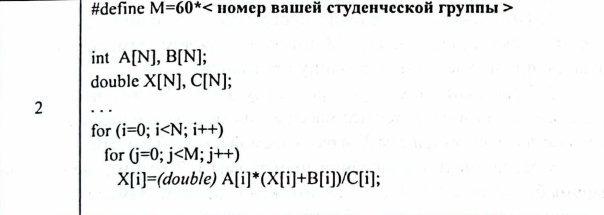
Проверил: Орлов Д.А., Хиль С.

Москва, 2024 г.

Вариант 2

**Задание**

Написать на языке С и отладить программу с использованием технологии CUDA и памяти global, shared и constant согласно варианту:

****

Параметры системы:  
RTX 2060, 1920 Cuda ядер, 30 мультипроцессоров, L1 кэш 64 КБ (у 1 МП), L2 кэш 3 МБ   
Код программы:

// Бригада №2, Вариант 2, ЛР3

#include <iostream>

#include <vector>

#include <fstream>

#include <chrono>

#include <cuda\_runtime.h>

#define N 512 // Размер массивов для пп. 3-7

#define M 60\*7 // Переменная из варианта для цикла

#define bigN 65536 // 64 KB

#define coeff 64 // Коэффициент деления для разбиения массивов

#define CAP (bigN / coeff) // По 1 КБ на 4 массива (Итого 4 КБ)

using namespace std;

using namespace std::chrono;

// Макрос для проверки CUDA

#define checkCUDA(expression) { gpuAssert((expression), \_\_FILE\_\_, \_\_LINE\_\_); }

inline void gpuAssert(cudaError\_t status, const char\* file, int line)

{

if (status != cudaSuccess)

{

std::cerr << "GPUassert: " << cudaGetErrorString(status) << ", " << file << ", " << line << std::endl;

std::exit(EXIT\_FAILURE);

}

}

// Функция для расчёта времени выполнения функции

template<typename Func>

double measureExecutionTime(Func func) {

auto startTime = high\_resolution\_clock::now();

func();

auto endTime = high\_resolution\_clock::now();

auto duration = duration\_cast<nanoseconds>(endTime - startTime).count();

return duration / 1e6; // Перевод в миллисекунды

}

int A[N], B[N];

double X[N], C[N];

int bigA[bigN], bigB[bigN];

double bigX[bigN], bigC[bigN];

void fillArrays(int \*a, int \*b, double \*c, double \*x) {

for (int i = 0; i < N; i++) {

a[i] = 2;

b[i] = 1;

c[i] = 1.5;

x[i] = 1.0;

}

}

void fillBigArrays(int \*a, int \*b, double \*c, double \*x) {

for (int i = 0; i < bigN; i++) {

a[i] = 2;

b[i] = 1;

c[i] = 1.5;

x[i] = 1.0;

}

}

void hostWork(const int \*a, const int \*b, const double \*c, double \*x) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < M; j++) {

x[i] = (double) a[i] \* (x[i] + b[j]) / c[j];

}

}

}

\_\_global\_\_ void globalMemoryGPU(const int \*a, const int \*b, const double \*c, double \*x, int size) {

uint8\_t i = threadIdx.x + blockIdx.x \* blockDim.x;

for (int y = 0; y < size; y++) {

int bigIndex = i \* size + y;

for (int j = 0; j < M; j++) {

x[bigIndex] = (double) a[bigIndex] \* (x[bigIndex] + b[bigIndex]) / c[bigIndex];

}

}

}

\_\_global\_\_ void sharedMemoryGPU(const int \*a, const int \*b, const double \*c, double \*x, int size) {

\_\_shared\_\_ double sharedX[N];

\_\_shared\_\_ int sharedA[N];

\_\_shared\_\_ int sharedB[N];

\_\_shared\_\_ double sharedC[N];

uint8\_t i = threadIdx.x + blockIdx.x \* blockDim.x;

// for (int y = i \* size; y < (i + 1) \* size; y++) {

for (int y = 0; y < size; y++) {

int bigIndex = i \* size + y;

// int bigIndex = y;

sharedA[bigIndex] = a[bigIndex];

sharedB[bigIndex] = b[bigIndex];

sharedC[bigIndex] = c[bigIndex];

sharedX[bigIndex] = x[bigIndex];

\_\_syncthreads();

for (int j = 0; j < M; j++) {

sharedX[bigIndex] = (double) sharedA[bigIndex] \* (sharedX[bigIndex] + sharedB[bigIndex]) / sharedC[bigIndex];

}

\_\_syncthreads();

x[bigIndex] = sharedX[bigIndex];

}

}

\_\_constant\_\_ int constantA[N];

\_\_constant\_\_ int constantB[N];

\_\_constant\_\_ double constantC[N];

\_\_global\_\_ void constantMemoryGPU(double \*x, int size) {

\_\_shared\_\_ double sharedX[N];

uint8\_t i = threadIdx.x + blockIdx.x \* blockDim.x;

for (int y = 0; y < size; y++) {

int bigIndex = i \* size + y;

sharedX[bigIndex] = x[bigIndex];

\_\_syncthreads();

for (int j = 0; j < M; j++) {

sharedX[bigIndex] = (double) constantA[bigIndex] \* (sharedX[bigIndex] + constantB[bigIndex]) / constantC[bigIndex];

}

\_\_syncthreads();

x[bigIndex] = sharedX[bigIndex];

}

}

\_\_global\_\_ void sharedMemoryCapGPU(const int \*a, const int \*b, const double \*c, double \*x, int size, int cap\_size) {

uint8\_t i = threadIdx.x + blockIdx.x \* blockDim.x;

for (int r = 0; r < coeff; r++) {

\_\_shared\_\_ double sharedX[CAP];

\_\_shared\_\_ int sharedA[CAP];

\_\_shared\_\_ int sharedB[CAP];

\_\_shared\_\_ double sharedC[CAP];

// for (int y = i \* cap\_size; y < (i + 1) \* cap\_size; y++) {

for (int y = 0; y < cap\_size; y++) {

int capIndex = i \* cap\_size + y;

// int capIndex = y;

int bigIndex = capIndex + r \* CAP;

// int bigIndex = y + r \* CAP;

sharedA[capIndex] = a[bigIndex];

sharedB[capIndex] = b[bigIndex];

sharedC[capIndex] = c[bigIndex];

sharedX[capIndex] = x[bigIndex];

\_\_syncthreads();

for (int j = 0; j < M; j++) {

sharedX[capIndex] = (double) sharedA[capIndex] \* (sharedX[capIndex] + sharedB[capIndex]) / sharedC[capIndex];

}

\_\_syncthreads();

x[bigIndex] = sharedX[capIndex];

}

}

}

\_\_constant\_\_ int constantBigA[CAP];

\_\_constant\_\_ int constantBigB[CAP];

\_\_constant\_\_ double constantBigC[CAP];

\_\_global\_\_ void constantMemoryCapGPU(double \*x, int size, int cap\_size, int cap\_index) {

uint8\_t i = threadIdx.x + blockIdx.x \* blockDim.x;

\_\_shared\_\_ double sharedX[CAP];

for (int y = 0; y < cap\_size; y++) {

int capIndex = i \* cap\_size + y;

int bigIndex = capIndex + cap\_index \* CAP;

sharedX[capIndex] = x[bigIndex];

\_\_syncthreads();

for (int j = 0; j < M; j++) {

sharedX[capIndex] = (double) constantBigA[capIndex] \* (sharedX[capIndex] + constantBigB[capIndex]) / constantBigC[capIndex];

}

\_\_syncthreads();

x[bigIndex] = sharedX[capIndex];

}

}

void runCUDA(const int GRID\_SIZE, const int BLOCK\_SIZE) {

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

int \*d\_A, \*d\_B;

double \*d\_C, \*d\_X;

const int forSize = N / (GRID\_SIZE \* BLOCK\_SIZE);

const int forBigSize = bigN / (GRID\_SIZE \* BLOCK\_SIZE);

const int forCapSize = CAP / (GRID\_SIZE \* BLOCK\_SIZE);

const dim3 block(BLOCK\_SIZE); // Размер блока (одномерный)

const dim3 grid(GRID\_SIZE); // Размер сетки из блоков (одномерная)

fillArrays(A, B, C, X);

cout << "\n-------------------------- GLOBAL PART STARTED --------------------------\n";

checkCUDA(cudaMalloc(&d\_A, N \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_B, N \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_C, N \* sizeof(double)));

checkCUDA(cudaMalloc(&d\_X, N \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_A, A, N \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_B, B, N \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_C, C, N \* sizeof(double), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_X, X, N \* sizeof(double), cudaMemcpyHostToDevice));

cudaEventRecord(start); // Запуск таймера

globalMemoryGPU <<< grid, block >>>(d\_A, d\_B, d\_C, d\_X, forSize); // Запуск процедуры

checkCUDA(cudaPeekAtLastError()); // Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(X, d\_X, N \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

float time; // Время

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream globalTime("../gpu\_global.csv", std::ios::app);// Открытие файла

cout << "\nProcessing time for global memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

globalTime << N << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

globalTime.close();// Закрытие файла

fillBigArrays(bigA, bigB, bigC, bigX);

checkCUDA(cudaMalloc(&d\_A, bigN \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_B, bigN \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_C, bigN \* sizeof(double)));

checkCUDA(cudaMalloc(&d\_X, bigN \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_A, bigA, bigN \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_B, bigB, bigN \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_C, bigC, bigN \* sizeof(double), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_X, bigX, bigN \* sizeof(double), cudaMemcpyHostToDevice));

cudaEventRecord(start); // Запуск таймера

globalMemoryGPU <<< grid, block >>>(d\_A, d\_B, d\_C, d\_X, forBigSize); // Запуск процедуры

checkCUDA(cudaPeekAtLastError()); // Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(bigX, d\_X, bigN \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

time; // Время

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream globalBigTime("../gpu\_global.csv", std::ios::app);// Открытие файла

cout << "\nProcessing time for global memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

globalBigTime << bigN << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

globalBigTime.close(); // Закрытие файла

cout << "\n-------------------------- GLOBAL PART ENDED --------------------------\n";

cudaDeviceSynchronize();

cout << "\n-------------------------- SHARED PART STARTED --------------------------\n";

fillArrays(A, B, C, X);

checkCUDA(cudaMalloc(&d\_A, N \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_B, N \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_C, N \* sizeof(double)));

checkCUDA(cudaMalloc(&d\_X, N \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_A, A, N \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_B, B, N \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_C, C, N \* sizeof(double), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_X, X, N \* sizeof(double), cudaMemcpyHostToDevice));

cudaEventRecord(start); // Запуск таймера

sharedMemoryGPU <<< grid, block >>>(d\_A, d\_B, d\_C, d\_X, forSize); // Запуск процедуры

checkCUDA(cudaPeekAtLastError());// Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(X, d\_X, N \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream sharedTime("../gpu\_shared.csv", std::ios::app); // Открытие файла

cout << "\nProcessing time for shared memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

sharedTime << N << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

sharedTime.close();// Закрытие файла

cout << "\n-------------------------- SHARED PART FINISHED --------------------------\n";

cudaDeviceSynchronize();

cout << "\n-------------------------- CONSTANT PART STARTED --------------------------\n";

fillArrays(A, B, C, X);

checkCUDA(cudaMalloc(&d\_X, N \* sizeof(double)));

checkCUDA(cudaMemcpyToSymbol(constantA, A, N \* sizeof(int)));

checkCUDA(cudaMemcpyToSymbol(constantB, B, N \* sizeof(int)));

checkCUDA(cudaMemcpyToSymbol(constantC, C, N \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_X, X, N \* sizeof(double), cudaMemcpyHostToDevice));

cudaEventRecord(start); // Запуск таймера

constantMemoryGPU <<< grid, block >>>(d\_X, forSize); // Запуск процедуры

checkCUDA(cudaPeekAtLastError());// Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(X, d\_X, N \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream constantTime("../gpu\_const.csv", std::ios::app);// Открытие файла

cout << "\nProcessing time for constant memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

constantTime << N << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

constantTime.close(); // Закрытие файла

cout << "\n-------------------------- CONSTANT PART ENDED --------------------------\n";

cudaDeviceSynchronize();

cout << "\n-------------------------- SHARED CAP PART STARTED --------------------------\n";

fillBigArrays(bigA, bigB, bigC, bigX);

checkCUDA(cudaMalloc(&d\_A, bigN \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_B, bigN \* sizeof(int)));

checkCUDA(cudaMalloc(&d\_C, bigN \* sizeof(double)));

checkCUDA(cudaMalloc(&d\_X, bigN \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_A, bigA, bigN \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_B, bigB, bigN \* sizeof(int), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_C, bigC, bigN \* sizeof(double), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_X, bigX, bigN \* sizeof(double), cudaMemcpyHostToDevice));

cudaEventRecord(start); // Запуск таймера

sharedMemoryCapGPU <<< grid, block >>>(d\_A, d\_B, d\_C, d\_X, forBigSize, forCapSize); // Запуск процедуры

checkCUDA(cudaPeekAtLastError());// Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(bigX, d\_X, bigN \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream capSharedTime("../gpu\_shared\_cap.csv", std::ios::app); // Открытие файла

cout << "\nProcessing time for shared cap memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

capSharedTime << bigN << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

capSharedTime.close(); // Закрытие файла

cout << "\n-------------------------- SHARED CAP PART ENDED --------------------------\n";

cudaDeviceSynchronize();

cout << "\n-------------------------- CONSTANT CAP PART STARTED --------------------------\n";

fillBigArrays(bigA, bigB, bigC, bigX);

checkCUDA(cudaMalloc(&d\_X, bigN \* sizeof(double)));

checkCUDA(cudaMemcpy(d\_X, bigX, bigN \* sizeof(double), cudaMemcpyHostToDevice));

/// CONSTANT CAP

cudaEventRecord(start); // Запуск таймера

for (int i = 0; i < coeff; ++i)

{

checkCUDA(cudaMemcpyToSymbol(constantBigA, bigA + coeff, CAP \* sizeof(int)));

checkCUDA(cudaMemcpyToSymbol(constantBigB, bigB + coeff, CAP \* sizeof(int)));

checkCUDA(cudaMemcpyToSymbol(constantBigC, bigC + coeff, CAP \* sizeof(double)));

constantMemoryCapGPU <<< grid, block >>>(d\_X, forBigSize, forCapSize, coeff); // Запуск процедуры

}

checkCUDA(cudaPeekAtLastError());// Проверка на ошибки

cudaEventRecord(stop); // Остановка таймера

checkCUDA(cudaMemcpy(X, d\_X, bigN \* sizeof(double), cudaMemcpyDeviceToHost));

cudaFree(d\_X);

cudaEventSynchronize(stop); // Синхронизация таймера

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

ofstream capConstantTime("../gpu\_const\_cap.csv", std::ios::app); // Открытие файла

cout << "\nProcessing time for constant cap memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

capConstantTime << bigN << "," << BLOCK\_SIZE << "," << GRID\_SIZE << "," << time << "\n";

capConstantTime.close(); // Закрытие файла

cout << "\n-------------------------- CONSTANT CAP PART ENDED --------------------------\n";

}

int main() {

cout << "\n-------------------------- PROCESSING STARTED --------------------------\n";

fillArrays(A, B, C, X);

cout << "\n-------------------------- CPU PART STARTED --------------------------\n";

double time = measureExecutionTime([&](){ hostWork(A, B, C, X);});

ofstream cpuTime("../cpu.csv", std::ios::app);

cout << "\nProcessing time for CPU (ms): " << time << "\n";

cpuTime << N << "," << time << "\n";

cpuTime.close();

cout << "\n-------------------------- CPU PART FINISHED --------------------------\n";

vector<int> block\_sizes = {1, 4, 32, 128, 256};

vector<int> grid\_sizes = {1, 2, 4};

cout << "\n-------------------------- CUDA PART STARTED --------------------------\n";

for (const auto &grid : grid\_sizes) {

for (const auto &block : block\_sizes) {

runCUDA(grid, block);

}

}

cout << "\n-------------------------- CUDA PART FINISHED --------------------------\n";

cout << "\n-------------------------- PROCESSING FINISHED --------------------------\n";

return 0;

}

Время выполнения вычислений на GPU при N=512

|  |  |  |  |
| --- | --- | --- | --- |
| **Число блоков/число потоков** | **Только глобальная память** | **Разделяемая память** | **Разделяемая и константная память** |
| 1/1 | 85.4188 | 53.5162 | 53.4952 |
| 1/4 | 15.3188 | 13.3958 | 13.5952 |
| 1/32 | 2.35846 | 2.06643 | 1.69165 |
| 1/256 | 0.77616 | 0.641024 | 0.633728 |
| 2/1 | 35.3137 | 26.3508 | 26.3496 |
| 2/4 | 7.48518 | 6.60419 | 6.60269 |
| 2/32 | 0.995776 | 0.84512 | 0.849568 |
| 2/256 | 0.409344 | 0.32816 | 0.326176 |
| 4/1 | 14.6047 | 13.1825 | 13.1724 |
| 4/4 | 3.76627 | 3.32595 | 3.31187 |
| 4/32 | 0.486848 | 0.432448 | 0.434112 |
| 4/256 | 0.0136 | 0.020096 | 0.015904 |

Время выполнения вычислений на GPU при N=65536

|  |  |  |  |
| --- | --- | --- | --- |
| **Число блоков/число потоков** | **Только глобальная память** | **Разделяемая память** | **Разделяемая и константная память** |
| 1/1 | 7759.72 | 6897.48 | 16356.9 |
| 1/4 | 1991.31 | 1769.42 | 4066.32 |
| 1/32 | 309.551 | 237.644 | 589.935 |
| 1/256 | 113.953 | 81.3798 | 218.426 |
| 2/1 | 3874.32 | 3555.83 | 8110.07 |
| 2/4 | 972.965 | 879.405 | 2037.7 |
| 2/32 | 139.463 | 118.339 | 296.972 |
| 2/256 | 60.7184 | 52.1145 | 88.7054 |
| 4/1 | 1903.3 | 1716.28 | 4041.58 |
| 4/4 | 502.616 | 455.1 | 1022.64 |
| 4/32 | 69.8035 | 53.3846 | 151.082 |
| 4/256 | 24.692 | 19.5544 | 47.353 |

Сравнительная таблица времён выполнения вычислений

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Число блоков / число потоков | | | | | | | | | | | | |
| N | Память | 1/1 | 1/4 | 1/32 | 1/128 | 2/1 | 2/4 | 2/32 | 2/128 | 4/1 | 4/4 | 4/32 | 4/128 |
| 512 | Glob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Shar | 1.60 | 1.14 | 1.14 | 1.21 | 1.34 | 1.13 | 1.18 | 1.25 | 1.11 | 1.13 | 1.13 | 0.68 |
| Shar& Const | 1.60 | 1.13 | 1.39 | 1.22 | 1.34 | 1.13 | 1.17 | 1.25 | 1.11 | 1.14 | 1.12 | 0.86 |
| 65536 | Glob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Shar | 1.13 | 1.13 | 1.30 | 1.40 | 1.09 | 1.11 | 1.18 | 1.17 | 1.11 | 1.10 | 1.31 | 1.26 |
| Shar& Const | 0.47 | 0.49 | 0.52 | 0.52 | 0.48 | 0.48 | 0.47 | 0.68 | 0.47 | 0.49 | 0.46 | 0.52 |