

```
//  
// GenefX64.cpp : Defines the entry point for the console application.  
//  
// Copyright (c) 2009 David Underbakke and Yves Gallot  
// Copyright (c) 2010,2011 Shoichiro Yamada  
// Minor changes by Ken Brazier 2010, 2011  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <time.h>  
#include <string.h>  
#include <signal.h>  
  
#include <cuda.h>  
#include <cuda_runtime.h>  
#include < cufft.h>  
#include < cutil_inline.h>  
  
#include <gmp.h>  
  
#define CPU_TARGET "CUDA"  
  
#ifndef M_PI  
#define M_PI 3.1415926535897932364626  
#endif  
  
#define SaveFileVersion 221  
#define ProgVersionText "2.2.1"  
  
#ifndef PROPMAJOR  
#define PROPMAJOR 1  
#endif  
#ifndef PROPMINOR  
#define PROPMINOR 3  
#endif  
  
//#define STRIDE 128  
//#define STRIDE 256  
//#define SHIFT 8  
  
typedef struct  
{  
    double x, y;  
} Complex;  
  
typedef int Int32;  
typedef unsigned int UInt32;  
typedef unsigned long long UInt64;  
  
static cufftHandle plan;  
static Complex * e1;  
static Complex * g_e1;  
static Complex * g_z;  
static Complex * g_zz;  
static double * g_f1;  
static double * g_f2;  
static float maxErr;
```

```

static float * g_maxErr;
static float * l_maxErr;
static int zSize;
int quitting;
static UInt32 * Na;

#define C5152 6755399441055744.0

/* optimize this code for your compiler and processor */
#   define round_x86(x) ((x >= 0) ? floor(x + 0.5) : ceil(x - 0.5))
#   define round(x) ((x + C5152) - C5152)
#   define B_MAX 30000000

#   define myAlloc(n) malloc(n)
#   define myFree(x) free(x)

float ErrThreshold; // error threshold

void write_checkpoint(UInt32 b, UInt32 m, UInt32 i, Complex *z, time_t startTime);
int read_checkpoint(UInt32 bNew, UInt32 mNew, Complex *z, time_t *startTime, UInt32
*iStart);
//1.05 void write_log(char *fmt, ...);
void write_log(char *buffer);

static UInt32 lpt(const UInt32 n) /* least power of two greater than n */
{
    UInt32 i;
    for (i = 1; i < n; i *= 2);
    return i;
}

static UInt32 ilog(const UInt32 n) /* floor log base 2 */
{
    UInt32 r, i;
    for (r = 0, i = n; i != 1; i /= 2, ++r);
    return r;
}

static unsigned int lg(const UInt32 * const a, const unsigned int len)
{
    UInt32 x;
    unsigned int l = (len - 1) * 8 * sizeof(UInt32) - 1;

    for (x = a[len-1]; x != 0; x /= 2)
        ++l;

    return l;
}

static UInt32 mul_1_add_n(UInt32 * const a, const UInt32 n, const UInt32 * const b, const
unsigned int len)
{
    unsigned int i;
    UInt64 l = 0;

    for (i = 0; i != len; ++i)
    {
        l += b[i] * (UInt64)n + a[i];
        a[i] = (UInt32)l;
    }
}

```

```

        l >= 32;
    }
    return (UInt32)l;
}
static void init_device(int device_number)
{
    int device_count=0;
    struct cudaDeviceProp properties;

    cudaGetDeviceCount( &device_count);
    if (device_number >= device_count)
    {
        printf("device_number >= device_count ... exiting\n");
        exit(2);
    }

    cudaSetDevice(device_number);
    cudaSetDeviceFlags(cudaDeviceBlockingSync);
    // From Iain
    cudaGetDeviceProperties(&properties, device_number);

    if (properties.major == PROPMAJOR && properties.minor < PROPMINOR){
        printf("A GPU with compute capability >= %d.%d is required for this program.
        \n", PROPMAJOR, PROPMINOR);
        exit(2);
    }
}

static void FFTfree(Complex * const z)
{
    cufftSafeCall(cufftDestroy(plan));
    myFree(z);
    myFree(e1);
    myFree(l_maxErr);
    cutilSafeCall(cudaFree((char *)g_z));
    cutilSafeCall(cudaFree((char *)g_zz));
    cutilSafeCall(cudaFree((char *)g_e1));
    cutilSafeCall(cudaFree((char *)g_f1));
    cutilSafeCall(cudaFree((char *)g_f2));
    cutilSafeCall(cudaFree((char *)g_maxErr));
}

#define BLOCK_DIM 16

// This kernel is optimized to ensure all global reads and writes are coalesced,
// and to avoid bank conflicts in shared memory. This kernel is up to 11x faster
// than the naive kernel below. Note that the shared memory array is sized to
// (BLOCK_DIM+1)*BLOCK_DIM. This pads each row of the 2D block in shared memory
// so that bank conflicts do not occur when threads address the array column-wise.
__global__ void transpose(double *odata, double *idata, int width, int height)
{
    __shared__ double block[BLOCK_DIM][BLOCK_DIM+1];

    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {

```

```

        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width))
    {
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}

// When doubles must be read or written in pairs, we use double2 so such accesses coalesce.
__global__ void mul_kernel(double2 *g_z)
{
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    double2 g_z_old = g_z[threadID];
    g_z[threadID] = make_double2(g_z_old.x * g_z_old.x - g_z_old.y * g_z_old.y,
    g_z_old.x * g_z_old.y * 2.0);
}

static void FFTsquareFFT(Complex * z, unsigned int n)
{
    cufftSafeCall(cufftExecZ2Z(plan, (cufftDoubleComplex *)g_z, (cufftDoubleComplex
    *)g_z, CUFFT_FORWARD));
    mul_kernel<<< n/128,128 >>> ((double2*)g_z);
    cufftSafeCall(cufftExecZ2Z(plan, (cufftDoubleComplex *)g_z, (cufftDoubleComplex
    *)g_z, CUFFT_INVERSE));
}

static Complex * FFTinitGFN(const UInt32 m, const double x, unsigned int * const rn1, int *
const rSHIFT)
{
    unsigned int n1, i;
    Complex * z;
    const double Pi_Twon = -M_PI / m;
    int STRIDE,SHIFT;

    n1 = m / 2;
    if(n1 > 2097152) SHIFT = 10;
    else if(n1 > 524288) SHIFT = 9;
        else SHIFT = 8;
    STRIDE = 1 << SHIFT;
    *rSHIFT = SHIFT;

    e1 = (Complex *)myAlloc((n1+1) * sizeof(Complex));

    cutilSafeCall(cudaMalloc((void**)&g_e1, sizeof(Complex)*(n1+STRIDE*STRIDE*2)));
    cutilSafeCall(cudaMalloc((void**)&g_zz, sizeof(Complex)*(n1+STRIDE*STRIDE*2)));

    for (i = 0; i != n1; ++i)
    {
        e1[i].x = cos(Pi_Twon * i);
        e1[i].y = sin(Pi_Twon * i);
    }
}

```

```

e1[n1].x = cos(0.0);
e1[n1].y = sin(0.0);

cutilSafeCall(cudaMemcpy(g_zz,e1,sizeof(Complex)*n1,cudaMemcpyHostToDevice));
dim3 grid(STRIDE*2/BLOCK_DIM,STRIDE*2/BLOCK_DIM, 1);
dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);
for(i=0;i<n1;i+=(STRIDE*STRIDE*2))
    transpose<<< grid, threads >>>((double *)&g_e1[i],(double *)&g_zz[i],(int)
        STRIDE*2,(int) STRIDE*2);

cufftSafeCall(cufftPlan1d(&plan, n1, CUFFT_ZZZ, 1));
//1.048 cufftSetCompatibilityMode(plan,CUFFT_COMPATIBILITY_NATIVE); //1.047

zSize = n1 * sizeof(Complex);
z = (Complex *)myAlloc(n1 * sizeof(Complex));
cutilSafeCall(cudaMalloc((void**)&g_z, sizeof(Complex)*(n1+STRIDE*STRIDE*2)));
cutilSafeCall(cudaMalloc((void**)&g_f1, sizeof(double)*(n1/STRIDE)));
cutilSafeCall(cudaMalloc((void**)&g_f2, sizeof(double)*(n1/STRIDE)));
cutilSafeCall(cudaMalloc((void**)&g_maxErr, sizeof(float))); //1.00

z[0].x = x; z[0].y = 0;
for (i = 1; i < n1 ; ++i)
    z[i].x = z[i].y = 0;
maxErr = 0;
l_maxErr = (float *)myAlloc(sizeof(float));
l_maxErr[0] = 0;
cutilSafeCall(cudaMemcpy(g_maxErr,l_maxErr,sizeof(float),cudaMemcpyHostToDevice));
//1.00
*rn1 = n1;
return z;
}

```

```

#define IDX(i)
(((i)>>(SHIFT*2+2))<<(SHIFT*2+2))+(((i)&((2<<SHIFT)*(2<<SHIFT)-1))>>(SHIFT+1))+(((i)&((2<<
SHIFT)-1))<<(SHIFT+1)))
__global__ void FFTnextStepGFN_kernel(double * g_z,double * g_e1,double * g_f1,double *
g_f2, Int32 base, double invBase, double t, unsigned int n1,float * g_maxErr, double tt,
const int SHIFT)
{
    const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int i;
    double x, y, xi, yi, f1, f2;
    float errx, erry,maxErr;
    double zx,zy,ex,ey;
    const int STRIDE = 1 << SHIFT;

    f1 = f2 = maxErr = 0;

    for (i = threadID*STRIDE; i != threadID*STRIDE+STRIDE; ++i)
    {
        zx = g_z[IDX((i)*2)];
        zy = - g_z[IDX((i)*2+1)];
        ex = g_e1[IDX((i)*2)];
        ey = g_e1[IDX((i)*2+1)];
        x = tt * (ex * zx + ey * zy);
        y = tt * (ey * zx - ex * zy);
        xi = round(x);
        yi = round(y);
    }
}

```

```

    errx = fabsf((float)(x - xi));
    if (errx > maxErr) maxErr = errx;
    erry = fabsf((float)(y - yi));
    if (erry > maxErr) maxErr = erry;

    xi = f1 + t * xi;
    yi = f2 + t * yi;

    x = xi * invBase;
    y = yi * invBase;
    f1 = round(x);
    f2 = round(y);

    xi -= f1 * base;
    yi -= f2 * base;

    g_z[IDX((i)*2)] = ex * xi + ey * yi;
    g_z[IDX((i)*2+1)] = ex * yi - ey * xi;
}
g_f1[threadID]=f1;
g_f2[threadID]=f2;
atomicMax((int*)g_maxErr, __float_as_int(maxErr));
}
__global__ void FFTnextStepGFN_kernel2(double * g_z,double * g_e1,double * g_f1,double *
g_f2, Int32 base, double invBase, double t, unsigned int n1,float * g_maxErr, const int
SHIFT)
{
    const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int i;
    double x, y, xi, yi, f1, f2;
    float errx, erry, maxErr;
    double ex,ey,zx,zy;
    const int STRIDE = 1 << SHIFT;

    maxErr = 0;

    if(threadID == 0)
    {
        f1=g_f1[n1/STRIDE-1];
        f2=g_f2[n1/STRIDE-1];
        f2 = -f2;          /* a[n] = -a[0] */
    }
    else
    {
        f2=g_f1[threadID-1];
        f1=g_f2[threadID-1];
    }

    for(i=threadID*STRIDE;i<threadID*STRIDE+8;++i) //0.99
    {

        zx = g_z[IDX(i*2)];
        zy = g_z[IDX(i*2+1)];
        ex = g_e1[IDX(i*2)];
        ey = g_e1[IDX(i*2+1)];
        x = f2 + ex * zx - ey * zy;
        y = f1 + ex * zy + ey * zx;
        xi = round(x);

```

```

        yi = round(y);

        errx = fabsf((float)(x - xi));
        if (errx > maxErr) maxErr = errx;
        erry = fabsf((float)(y - yi));
        if (erry > maxErr) maxErr = erry;

        x = xi * invBase;
        y = yi * invBase;
        f2 = round(x);
        f1 = round(y);
        xi -= f2 * base;
        yi -= f1 * base;

        zx = ex * xi + ey * yi;
        zy = ex * yi - ey * xi;
        g_z[IDX(i*2)]=zx;
        g_z[IDX(i*2+1)]=zy;
    }

    zx = g_z[IDX(i*2)];
    zy = g_z[IDX(i*2+1)];
    ex = g_e1[IDX(i*2)];
    ey = g_e1[IDX(i*2+1)];
    x = ex * zx - ey * zy;
    y = ex * zy + ey * zx;
    xi = round(x);
    yi = round(y);
    errx = fabsf((float)(x - xi));
    if (errx > maxErr) maxErr = errx;
    erry = fabsf((float)(y - yi));
    if (erry > maxErr) maxErr = erry;
    f2 += xi;
    f1 += yi;
    zx = ex * f2 + ey * f1;
    zy = ex * f1 - ey * f2;
    g_z[IDX(i*2)]=zx;
    g_z[IDX(i*2+1)]=zy;
    atomicMax((int*)g_maxErr, __float_as_int(maxErr));
}

```

```

static void FFTnextStepGFN(const Int32 base, const double invBase, const double t, const
unsigned int n1,const double tt,int const ithreads, const int SHIFT)
{
    int j;
    const int STRIDE = 1 << SHIFT;
    dim3 grid(STRIDE*2/BLOCK_DIM,STRIDE*2/BLOCK_DIM, 1);
    dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);

```

```

        for(j=0;j<n1;j+=(STRIDE*STRIDE*2))
            transpose<<< grid, threads >>>((double *)&g_zz[j],(double
            *)&g_z[j],(int) STRIDE*2,(int) STRIDE*2);
        FFTnextStepGFN_kernel<<< n1/STRIDE/ithreads,ithreads >>>((double *)g_zz,
        (double *)g_e1,g_f1,g_f2,base,invBase,t,n1,g_maxErr,tt,SHIFT);
        FFTnextStepGFN_kernel2<<< n1/STRIDE/ithreads,ithreads >>>((double *)g_zz,
        (double *)g_e1,g_f1,g_f2,base,invBase,t,n1,g_maxErr,SHIFT);
        for(j=0;j<n1;j+=(STRIDE*STRIDE*2))
            transpose<<< grid, threads >>>((double *)&g_z[j],(double
            *)&g_zz[j],(int) STRIDE*2,(int) STRIDE*2);

```

```
}
```

```
static void FFTgetResult(const Complex * const z, const Int32 base, Int32 * const a, const unsigned int n1)
```

```
{
    unsigned int i, k;
    const Complex * zPtr, * eiPtr, * ejPtr;
    const unsigned int n = n1;
    double c, s, x, y, xi, yi, f1, f2;
    const double t = 1.0 / n, invBase = 1.0 / base;
    float errx, erry;

    k = 0;
    f1 = f2 = 0;
    zPtr = z;
    ejPtr = &el[n1];
    eiPtr = el;
    for (i = n1; i != 0; --i)
    {
        c = ejPtr->x * eiPtr->x - ejPtr->y * eiPtr->y;
        s = ejPtr->x * eiPtr->y + ejPtr->y * eiPtr->x;
        ++eiPtr;

        x = f1 + t * (c * zPtr->x - s * zPtr->y);
        y = f2 + t * (s * zPtr->x + c * zPtr->y);
        ++zPtr;
        xi = round_x86(x);
        yi = round_x86(y);

        errx = fabsf((float)(x - xi));
        if (errx > maxErr) maxErr = errx;
        erry = fabsf((float)(y - yi));
        if (erry > maxErr) maxErr = erry;

        x = xi * invBase;
        y = yi * invBase;
        f1 = round_x86(x);
        f2 = round_x86(y);
        x = xi - f1 * base;
        y = yi - f2 * base;
        xi = round_x86(x);
        yi = round_x86(y);
        a[k] = (Int32)xi;
        a[k + n] = (Int32)yi;
        ++k;
    }

    f2 = -f2;          /* a[n] = -a[0] */

    for (k = 0; k != 9; ++k)
    {
        yi = f2 + a[k];
        xi = f1 + a[k + n];
        y = yi * invBase;
        x = xi * invBase;
        f2 = round_x86(y);
        f1 = round_x86(x);
        a[k] = (Int32)(yi - f2 * base);
    }
}
```



```

        a[k + n] = (Int32)(xi - f1 * base);
    }
    a[9] += (Int32)f2;
    a[9 + n] += (Int32)f1;
}

static int isOne(Int32 * const a, const unsigned int len, const Int32 base)
{
    unsigned int i;
    Int32 f, r;
    int rt;

    do
    {
        f = 0;
        for (i = 0; i != len; ++i)
        {
            f += a[i];
            r = f % base;
            if (r < 0) r += base;
            a[i] = r;
            f -= r;
            f /= base;
        }
        a[0] -= f;      /* a[n] = -a[0] */
    } while (f != 0);

    rt = 1;
    for (i = 1; i != len; ++i)
    {
        if (a[i] != 0)
        {
            rt = 0;
            break;
        }
    }

    if (a[0] != 1)
        rt = 0;

    return rt;
}

void SetQuitting(int sig)
{
    quitting = 1;
    printf("^C caught. Writing checkpoint.\n");
}

#ifdef WIN32
BOOL WINAPI HandlerRoutine(DWORD /*dwCtrlType*/)
{
    SetQuitting(1);
    return true;
}
#endif

static void check(const Int32 b, const UInt32 m, char *expectedResidue)
{
    time_t    startTime;

```

```

    UInt32 j, dgts;
    unsigned int i, Nlen, bt, n1, iStart;
    UInt32 * a;
    Int32 * Ra;
    Complex * z;
    int r;
    const double t1 = 1.0, t2 = 2.0, t3 = 2.0 / m;
    FILE * fp;
    char str1[132], str2[100], residue[30];
    const double invBase = 1.0 / b;
    long hours, minutes, seconds;
    int SHIFT;

    time_t ltime;           // used for time printout
    struct tm *today;       // used for time printout

#ifdef WIN32
    SetConsoleCtrlHandler(HandlerRoutine, TRUE);
#else
    // We log to file in most cases anyway.
    signal(SIGTERM, SetQuitting);
    signal(SIGINT, SetQuitting);
#endif

    time(&ltime);
    today = localtime(&ltime);

    if (b & 1)
    {
        fprintf(stderr, "Cannot test.  b is odd, thus b^m+1 is odd.\n");
        return;
    }

    j = m;
    while (j > 0)
    {
        if (j > 1 && j & 1)
        {
            fprintf(stderr, "Cannot test.  m is not a power of 2.\n");
            return;
        }
        j >= 1;
    }

    printf("Testing %d^%u+1...\r", b, m);

    startTime = time(NULL);

    int ithreads;

    {
        mpz_t m_Na;
        int m_b;
        size_t m_Na_size;
        int m_Na_size_byte;
        long int *m_a;
        unsigned int *m_a_32;
        int m_a_32_len;
        m_b = b;
    }

```

```

    mpz_init_set_ui(m_Na,m_b);
    for(j = m; j != 1; j/=2)
        mpz_mul(m_Na,m_Na,m_Na);
    m_Na_size = mpz_size(m_Na);
    m_Na_size_byte = m_Na_size*sizeof(long int);
    m_a = (long int*) malloc(m_Na_size*sizeof(long int));
    mpz_export(m_a,NULL,0,m_Na_size_byte,0,0,m_Na);
    m_a_32 = (unsigned int*) malloc(m_Na_size*sizeof(int)*2);
    for(j = 0; j < m_Na_size; j++)
    {
        m_a_32[j*2]=m_a[j] & 0xFFFFFFFF;
        m_a_32[j*2+1]=m_a[j] >> 32;
    }
    m_a_32_len = m_Na_size*2;
    if(m_a_32[m_a_32_len-1]== 0) m_a_32_len--;
    Na = m_a_32 ;
    Nlen = m_a_32_len;
    mpz_clear(m_Na);
    free(m_a);
}

z = FFTinitGFN(m, 2.0, &n1,&SHIFT);
iStart = lg(Na, Nlen) - 1;
if (read_checkpoint(b, m, z, &startTime, &iStart))
    printf("\rResuming %d^%u+1 from a checkpoint (%d iterations left)\n", b, m,
iStart); //1.00
else
    printf("\rTesting %d^%u+1...  %d steps to go ", b, m, iStart);
    fflush(stdout);

cutilSafeCall(cudaMemcpy(g_z,z,sizeof(Complex)*n1,cudaMemcpyHostToDevice));
if(n1 > 16384) ithreads = 128;
else ithreads = 16;
for (i = iStart; i != 0; --i)
{
    FFTsquareFFT(z, n1);
    bt = Na[i / (8 * sizeof(UInt32))] >> (i % (8 * sizeof(UInt32))) & 1;
    FFTnextStepGFN(b,invBase,(bt == 0) ? t1 : t2,n1,t3,ithreads,SHIFT);
    if (!(i & 0xffff))
    {
        printf("\rTesting %d^%u+1...  %d steps to go ", b, m, i);
        fflush(stdout);

        cutilSafeCall(cudaMemcpy(l_maxErr,g_maxErr,sizeof(float),cudaMemcpy
DeviceToHost)); //1.00
        if(l_maxErr[0]>maxErr)maxErr=l_maxErr[0]; //1.00

        if (maxErr > 0.45)
        {
            printf("\nmaxErr exceeded for %d^%u+1, %.4f > %.4f\n", b,
m, maxErr, 0.45);
            fp = fopen("genefer.log", "a");
            fprintf(fp, "maxErr exceeded for %d^%u+1, %.4f > %.4f\n",
b, m, maxErr, 0.45);
            fclose(fp);
            exit(0);
        }
    }

    cutilSafeCall(cudaMemcpy(z,g_z,sizeof(Complex)*n1,cudaMemcpyDeviceT

```

```

        oHost));
        write_checkpoint(b, m, i, z, startTime);
    }
    if (quitting)
    {
        cutilSafeCall(cudaMemcpy(z, g_z, sizeof(Complex)*n1, cudaMemcpyDeviceToHost));
        write_checkpoint(b, m, i, z, startTime);
        exit(0);
    }
}
FFTsquareFFT(z, n1);
cutilSafeCall(cudaMemcpy(z, g_z, sizeof(Complex)*n1, cudaMemcpyDeviceToHost));
cutilSafeCall(cudaMemcpy(l_maxErr, g_maxErr, sizeof(float), cudaMemcpyDeviceToHost));
//0.99
if(l_maxErr[0]>maxErr)maxErr=l_maxErr[0]; //0.99

Ra = (Int32 *)myAlloc(m * sizeof(UInt32));
FFTgetResult(z, b, Ra, n1);
FFTfree(z);

r = isOne(Ra, m, b);

dgts = (unsigned int)(m * log((double)b) / log(10.0)) + 1;

seconds = (long)(time(NULL) - startTime);
minutes = seconds / 60;
seconds = seconds - (minutes * 60);
hours = minutes / 60;
minutes = minutes - (hours * 60);

// calculate time
time(&lttime);
today = localtime(&lttime);

if (r == 1)
{
    // extra spaces to wipe out the steps to go message
    sprintf(str1, "\r%d^u+1 is a probable prime.          ", b, m);

    fp = fopen("verif.txt", "a");
    if (fp == NULL)
    {
        fprintf(stderr, "Cannot write results to verif.txt\n");
        exit(1);
    }
    fprintf(fp, "%d^u+1\n", b, m);
    fclose(fp);
}
else
{
    sprintf(residue, "%02x%02x%02x%02x%02x%02x%02x",
        (Ra[m-8] & 0xff), (Ra[m-7] & 0xff), (Ra[m-6] & 0xff), (Ra[m-5] & 0xff),
        (Ra[m-4] & 0xff), (Ra[m-3] & 0xff), (Ra[m-2] & 0xff), (Ra[m-1] & 0xff));
}

```

```

        if (2*log((double)b)/log(2.0) + ilog(m) + log((double)ilog(m))/log(2.0) <
53)
            sprintf(str1, "\r%d^u+1 is composite. (RES=%s)", b, m, residue);
        else
            sprintf(str1, "\r%d^u+1 is a probable composite. (RES=%s)", b, m,
residue);
    }

    sprintf(str2, " (%d digits) (err = %.4f) (time = %ld:%02ld:%02ld) %.8s\n", dgts,
maxErr, hours, minutes, seconds, asctime(today)+11); //1.045

    myFree(Ra);
    printf("%s\n%s", str1, str2);
    fp = fopen("genefer.log", "a");
    if (fp == NULL)
    {
        fprintf(stderr, "Cannot write results to genefer.log\n");
        exit(1);
    }
    fprintf(fp, "%s %s", str1+1, str2);
    fclose(fp);

    remove("genefer.ckpt");

    if (expectedResidue && strcmp(residue, expectedResidue))
        printf("Expected residue [%s] does not match actual residue [%s]\n",
expectedResidue, residue);

    return;
    //return r;
}

static void test()
{
    check(2030234, 8192, 0);
    check(1651902, 16384, 0);
    check(1277444, 32768, 0);
    check(857678, 65536, 0);
    check(572186, 131072, 0);
    check(24518, 262144, 0);
    check(75898, 524288, 0); //0.99
}

static double benchGFN(const Int32 b, const UInt32 n, const unsigned int m)
{
    unsigned int i, n1;
    clock_t clock0;
    double et;
    Complex * z;
    Int32 * Ra;
    const double t1 = 1.0, t2 = 2.0, t3 = 2.0 / n;
    const double invBase = 1.0 / b;
    int SHIFT;

    int ithreads;

    z = FFTinitGFN(n, (double)(b - 2), &n1, &SHIFT);
    cutilSafeCall(cudaMemcpy(g_z, z, sizeof(Complex)*n1, cudaMemcpyHostToDevice));
    if(n1 > 16384) ithreads = 128;

```

```

else ithreads = 16;
for (i = 0; i <= ilog(n); ++i)
{
    FFTsquareFFT(z, n1);
    FFTnextStepGFN(b, invBase, t1, n1, t3, ithreads, SHIFT);
}

clock0 = clock();
for (i = 0; i < m; ++i)
{
    FFTsquareFFT(z, n1);
    FFTnextStepGFN(b, invBase, (i % 2 == 0) ? t1 : t2, n1, t3, ithreads, SHIFT);
}
et = (clock() - clock0) / (double)CLOCKS_PER_SEC;

FTTsquareFFT(z, n1);
cutilSafeCall(cudaMemcpy(l_maxErr, g_maxErr, sizeof(float), cudaMemcpyDeviceToHost));
//1.00
if(l_maxErr[0]>maxErr)maxErr=l_maxErr[0]; //1.00
Ra = (Int32 *)myAlloc(n * sizeof(UInt32));
FFTgetResult(z, b, Ra, n1);

FFTfree(z);
myFree(Ra);

return et;
}

static void runBench()
{
    unsigned int i, m, dgts;
    Int32 b;
    UInt32 n;
    double et, etn;
    FILE * fp;

    fp = fopen("genefer.bench", "w");
    printf("Generalized Fermat Number Bench\n");
    if (fp != NULL)
        fprintf(fp, "Generalized Fermat Number Bench\n");

    for (i = 5; i < 16; ++i) //0.99
    {
        n = 1 << (i + 8);
        //1.047 m = 1 << (22 - i);
        m = 1 << (20 - (i >> 1)); //1.047
        b = (Int32)(B_MAX / pow(n, 0.3)) & 0xfffffffffe;
        et = benchGFN(b, n, m);
        etn = et / m;
        dgts = (unsigned int)(n * log((double)b) / log(10.0)) + 1;
        if (etn >= 1)
        {
            printf("%d^u+1\tTime: %.03g s/mul.\tErr: %.2e\t%u digits\n", b, n,
                etn, maxErr, dgts);
            if (fp != NULL)
                fprintf(fp, "%d^u+1\tTime: %.03g s/mul.\tErr: %.2e\t%u
                    digits\n", b, n, etn, maxErr, dgts);
        }
        else if (etn >= 1e-3)
    }
}

```

```

        {
            printf("%d^%u+1\tTime: %.03g ms/mul.\tErr: %.2e\t%u digits\n", b,
n, etn * 1e3, maxErr, dgts);
            if (fp != NULL)
                fprintf(fp, "%d^%u+1\tTime: %.03g ms/mul.\tErr: %.2e\t%u
digits\n", b, n, etn * 1e3, maxErr, dgts);
        }
        else
        {
            printf("%d^%u+1\tTime: %.03g us/mul.\tErr: %.2e\t%u digits\n", b,
n, etn * 1e6, maxErr, dgts);
            if (fp != NULL)
                fprintf(fp, "%d^%u+1\tTime: %.03g us/mul.\tErr: %.2e\t%u
digits\n", b, n, etn * 1e6, maxErr, dgts);
        }
    }

    if (fp != NULL)
        fclose(fp);
}

```

```

static void test_residue()
{
    check(230234, 8192, "5972a302c255e081");
    check(151902, 16384, "0a1a98e69b17f1e9");
    check(177444, 32768, "f69cc3e2334a43a8");
    check(157476, 65536, "9f64b3f0d545615c");
    check(52186, 131072, "1b196d6c0e4d778f");
    check(70000, 262144, "fa15b4b858fd2ff0");
    check(60000, 524288, "d34a021bbc70540a"); //0.99
    check(50000, 1048576, "41383f3c58852802"); //0.99
    check(4000, 2097152, "ff3daf8908789696"); //1.048 from AG5BPilot
    check(1248, 4194304, "8f985a974820a6d3"); //1.061 from AG5BPilot
    //0.99 check(2000, 8388608, "0000000000000000"); //0.99
}

```

```

static float check_limit(Int32 b, UInt32 n, UInt32 loops)
{
    unsigned int i, n1;
    Complex * z;
    Int32 * Ra;
    const double t1 = 1.0, t3 = 2.0 / n;
    const double invBase = 1.0 / b;
    int SHIFT;

    int ithreads;

    z = FFTinitGFN(n, (double)(b - 2), &n1, &SHIFT);
    cutilSafeCall(cudaMemcpy(g_z, z, sizeof(Complex)*n1, cudaMemcpyHostToDevice));

    if(n1 > 16384) ithreads = 128;
    else ithreads = 16;

    for (i = 0; i <= ilog(n); ++i)
    {
        FFTsquareFFT(z, n1);
        // always force a single direction to hit extreme
        FFTnextStepGFN(b, invBase, t1, n1, t3, ithreads, SHIFT);
    }
}

```

```

        cutilSafeCall(cudaMemcpy(l_maxErr,g_maxErr,sizeof(float),cudaMemcpyDeviceTo
Host)); //1.00
        if(l_maxErr[0]>maxErr)maxErr=l_maxErr[0]; //1.00

        if (maxErr > ErrThreshold)
        {
            FFTfree(z);

            return maxErr;
        }
    }

    for (i = 0; i < loops; ++i)
    {
        FFTsquareFFT(z, n1);
        // oscillate directions
        FFTnextStepGFN(b,invBase,t1,n1,t3,ithreads,SHIFT);

        cutilSafeCall(cudaMemcpy(l_maxErr,g_maxErr,sizeof(float),cudaMemcpyDeviceTo
Host)); //1.00
        if(l_maxErr[0]>maxErr)maxErr=l_maxErr[0]; //1.00
        if (maxErr > ErrThreshold)
        {
            FFTfree(z);

            return maxErr;
        }
    }
    FFTsquareFFT(z, n1);
    cutilSafeCall(cudaMemcpy(z,g_z,sizeof(Complex)*n1,cudaMemcpyDeviceToHost));

    Ra = (Int32 *)myAlloc(n * sizeof(UInt32));
    FFTgetResult(z, b, Ra, n1);

    FFTfree(z);

    myFree(Ra);

    return maxErr;
}
static void find_limits()
{
    // n here is really the same as m in other sections
    Int32 i, b, n, bStart, loops;
    float   maxError;

    Int32   lastErrorB;           // last error b value
    Int32   last5B;              // last b value with an error of 5.0
    float   lastErr;             // last error value

    FILE * fp;

    fp = fopen("genefer.limits", "w");
    printf("Generalized Fermat Number b Limits\n");
    if (fp != NULL)
        fprintf(fp, "Generalized Fermat Number b Limits\n");

    for (i = 5; i < 16; ++i) //0.99

```



```

{
    n = 1 << (i + 8);
    b = (Int32)(50000000 / pow(n, 0.3)) & 0xfffffffffe;
    loops = 1 << (16 - i);

    b = (b / 10000) * 10000;
    bStart = b;

    lastErrorB = 0;
    last5B = 0;
    lastErr = 0.0;

    ErrThreshold = (float) (0.290 + (double)i/600.0);

    do
    {
        b -= 5000;
        printf(" Checking limit for m = %d. Testing b = %d\r", n, b);
        fflush(stdout);

        maxError = check_limit(b, n, loops);

        if (maxError > ErrThreshold)
        {
            lastErrorB = b;
            lastErr = maxError;
            if (maxError > 0.499)
                last5B = b;
        }

    } while (maxError > ErrThreshold);

    printf("The upper bound m = %d, b = %d, Err = %.4f\n", n, b, maxError);
    printf(" Starting b = %d, Err b = %d, Err = %.4f, 5 Err b = %d\n", bStart,
lastErrorB, lastErr, last5B);
    fprintf(fp, "The upper bound m = %d, b = %d, Err = %.4f\n", n, b,
maxError);
    fprintf(fp, " Starting b = %d, Err b = %d, Err = %.4f, 5 Err b = %d\n",
bStart, lastErrorB, lastErr, last5B);
}
fclose(fp);
}

int main(int argc, char * argv[])
{
    int r, i, currentLine;
    UInt32 N;
    Int32 b;
    FILE * fp, * fpi, * fplog;
    char str[132];
    char str1[132], end[10];
    int device_number = 0;

    time_t ltime;           // used for time printout
    struct tm *today;       // used for time printout

    time(&ltime);
    today = localtime(&ltime);

```

```

quitting = 0;

//if ((nRet = SaveE02(e0, e1, e2)) != 0)
//{
//    printf("SaveE02 returned %d\n\n", nRet);
//    exit(4);
//}

printf("GeneferCUDA %s (%s) based on Genefer v2.2.1\n", ProgVersionText,
CPU_TARGET);
printf("Copyright (C) 2001-2003, Yves Gallot (v1.3)\n");
printf("Copyright (C) 2009, 2011 Mark Rodenkirch, David Underbakke (v2.2.1)\n");
printf("Copyright (C) 2010, 2011, Shoichiro Yamada (CUDA)\n");
if (argc > 1)
    if (strcmp(argv[1], "-v") == 0)
        return 0;
if (argc > 1)
    if (strcmp(argv[1], "-V") == 0)
        return 0;
printf("A program for finding large probable generalized Fermat primes.\n\n");
if (argc == 1)
{
    printf("Usage: GeneferCUDA [-d N] -b          run bench\n");
    printf("      GeneferCUDA [-d N] -t          run test\n");
    printf("      GeneferCUDA [-d N] -r          run residue test\n");
    printf("      GeneferCUDA [-d N] -q          test quick expression\n");
    printf("      GeneferCUDA [-d N] <filename> test <filename>\n");
    printf("      -d N set device number=N(default 0)\n");
    printf("      GeneferCUDA          use interactive mode\n\n");
    printf("  1. bench\n  2. test\n  3. test residue\n  4. normal\n");

    fgets(str, 80, stdin);
    r = atoi(str);
    if (r == 1)
    {
        printf("device number: ");
        fgets(str, 80, stdin);
        device_number = atoi(str);
        init_device(device_number);
        runBench();
        return 0;
    }
    if (r == 2)
    {
        printf("device number: ");
        fgets(str, 80, stdin);
        device_number = atoi(str);
        init_device(device_number);
        test();
        return 0;
    }
    if (r == 3)
    {
        printf("device number: ");
        fgets(str, 80, stdin);
        device_number = atoi(str);
        init_device(device_number);
        test_residue();
    }
}

```

```

        return 0;
    }

    printf("device number: ");
    fgets(str, 80, stdin);
    device_number = atoi(str);
    init_device(device_number);

    printf("N: ");
    fgets(str, 80, stdin);
    N = lpt(atoi(str));
    if (N < 8192) N = 8192;

    printf("b: ");
    fgets(str, 80, stdin);
    b = atoi(str) & 0xfffffffffe;

    check(b, N, 0);
    return 0;
}
if (strcmp(argv[1], "-d") == 0)
{
    if (argc < 3)
    {
        fprintf(stderr, "-d requires an expression\n");
        exit(2);
    }
    sprintf(str1, "%send", argv[2]);

    if (sscanf(str1, "%u%s", &device_number, &end) != 2)
    {
        fprintf(stderr, "-d requires an expression\n");
        exit(2);
    }
    init_device(device_number);
    argv += 2;
    argc -= 2;
}
else
    init_device(device_number); //1.045

if (strcmp(argv[1], "-b") == 0)
{
    runBench();
    return 0;
}
if (strcmp(argv[1], "-t") == 0)
{
    test();
    return 0;
}

if (strcmp(argv[1], "-r") == 0)
{
    test_residue();
    return 0;
}

```

```
if (strcmp(argv[1], "-l") == 0)
{
    find_limits();
    return 0;
}

if (strcmp(argv[1], "-q") == 0)
{
    if (argc < 3)
    {
        fprintf(stderr, "-q requires an expression\n");
        exit(2);
    }
    sprintf(str1, "%send", argv[2]);

    if (sscanf(str1, "%u^%u+1%s", &b, &N, &end) != 3)
    {
        fprintf(stderr, "\nExpression must be in b^m+1 form\n");
        exit(2);
    }
    check(b, N, 0);

    return 0;
}
fp = fopen(argv[1], "r");
if (fp == NULL)
{
    fprintf(stderr, "Cannot open '%s'\n", argv[1]);
    return 1;
}

fpi = fopen("genefer.ini", "r");
if (fpi != NULL)
{
    fgets(str, 132, fpi);
    currentLine = atoi(str);
    fclose(fpi);
    printf("Continue test of file '%s' at line %d - %.8s\n", argv[1],
currentLine, asctime(today)+11);
    fplog = fopen("genefer.log", "a");
    if (fp != NULL)
    {
        fprintf(fplog, "Continue test of file '%s' at line %d - %.8s\n",
argv[1], currentLine, asctime(today)+11);
        fclose(fplog);
    }
}
else
{
    currentLine = 0;
    printf("Start test of file '%s' - %.8s\n", argv[1], asctime(today)+11);
    fplog = fopen("genefer.log", "a");
    if (fp != NULL)
    {
        fprintf(fplog, "Start test of file '%s' - %.8s\n", argv[1],
asctime(today)+11);
        fclose(fplog);
    }
}
```

```

    for (i = 0; i < currentLine; ++i)
        fgets(str, 132, fp);

    while (fgets(str, 132, fp) != NULL)
    {
        if (sscanf(str, "%u%u", &b, &N) == 2)
        {
            if ((b != 0) && (b <= 0x7fffffff) && (N >= 16))
            {
                check(b, N, 0);
            }
        }
        ++currentLine;
        fpi = fopen("genefer.ini", "w");
        if (fpi != NULL)
        {
            fprintf(fpi, "%d\n", currentLine);
            fclose(fpi);
        }
    }

    fclose(fp);
    return 0;
}

```

*// Write a checkpoint file that we can read later so that the  
// test doesn't need to start from the beginning*

```

void write_checkpoint(UInt32 b, UInt32 m, UInt32 i, Complex *z, time_t startTime)
{
    FILE *fPtr;
    UInt32 SaveVer = SaveFileVersion, bytes;
    time_t nowTime = time(NULL) - startTime;

    fPtr = fopen("genefer.ckpt", "wb");
    if (!fPtr)
        return;

    fwrite(&SaveVer, 1, sizeof(SaveVer), fPtr);
    bytes = strlen(CPU_TARGET);
    fwrite(&bytes, 1, sizeof(bytes), fPtr);
    fwrite(CPU_TARGET, 1, bytes, fPtr);
    fwrite(&b, 1, sizeof(b), fPtr);
    fwrite(&m, 1, sizeof(m), fPtr);
    fwrite(&i, 1, sizeof(i), fPtr);
    fwrite(z, 1, zSize, fPtr);
    fwrite(&nowTime, 1, sizeof(nowTime), fPtr);

    fclose(fPtr);
}

```

```

int read_checkpoint(UInt32 bNew, UInt32 mNew, Complex *z, time_t *startTime, UInt32
*iStart)
{
    FILE *fPtr;
    UInt32 oldSaveVer = SaveFileVersion;
    UInt32 i, b, m, bytes, j;
    time_t usedTime;
    char build[50];

```

```

char    buffer[2000]; //1.05
Complex *zNew;

fPtr = fopen("genefer.ckpt", "rb");
if (!fPtr)
    return 0;

// check version
if (fread(&oldSaveVer, 1, sizeof(oldSaveVer), fPtr) != sizeof(oldSaveVer) ||
    oldSaveVer != SaveFileVersion)
{
    fprintf(stderr, "\nThe checkpoint version doesn't match current test. Current test
will be restarted\n");
    fclose(fPtr);
    return 0;
}

fread(&bytes, 1, sizeof(bytes), fPtr);
if (bytes != strlen(CPU_TARGET))
{
    sprintf(buffer, "Checkpoint saved by another build of genefer (%d %d). This version
cannot restart with it.", bytes, strlen(CPU_TARGET)); //1.05
    write_log(buffer); //1.05
    exit(0);
}

fread(build, 1, bytes, fPtr);
//1.049 if (!strcmp(build, CPU_TARGET))
build[bytes] = 0; //1.049 from AG5BPilot
if (strcmp(build, CPU_TARGET)) //1.049
{
    printf("Checkpoint saved by genefer %s. %s This version cannot restart with it",
build, CPU_TARGET);
    sprintf(buffer, "Checkpoint saved by genefer %s. This version cannot restart with
it", build); //1.05
    write_log(buffer); //1.05
    exit(0);
}

// check parameters
if (fread(&b, 1, sizeof(b), fPtr) != sizeof(b) ||
    fread(&m, 1, sizeof(m), fPtr) != sizeof(m) ||
    fread(&i, 1, sizeof(i), fPtr) != sizeof(i))
{
    fprintf(stderr, "\nThe checkpoint doesn't match current test. Current test will be
restarted\n");
    fclose(fPtr);
    return 0;
}

if (b != bNew || m != mNew)
{
    fprintf(stderr, "\nThe checkpoint doesn't match current test. Current test will be
restarted\n");
    fclose(fPtr);
    return 0;
}

// check for successful read of z, delayed until here since zSize can vary

```

```
zNew = (Complex *)myAlloc(zSize);
if (fread(zNew, 1, zSize, fPtr) != zSize ||
    fread(&usedTime, 1, sizeof(usedTime), fPtr) != sizeof(usedTime))
{
    fprintf(stderr, "\nThe checkpoint doesn't match current test. Current test will be
restarted\n");
    fclose(fPtr);
    myFree(zNew);
    return 0;
}

// have good stuff, do checkpoint
fclose(fPtr);
memcpy(z, zNew, zSize);
myFree(zNew);
*iStart = i-1;
*startTime = *startTime - usedTime;
return 1;
}

void write_log(char *buffer)
{
    va_list args;
    FILE *fp;

    printf("%s\n", buffer);

    fp = fopen("genefer.log", "a");
    if (fp == NULL)
    {
        fprintf(stderr, "Cannot write results to genefer.log\n");
        exit(1);
    }

    fprintf(fp, "%s\n", buffer);
    fclose(fp);
}
```