Calcul Parallèle et CUDA

http://www.ann.jussieu.fr/pironneau

Olivier Pironneau[1]

[1]University of Paris VI, Laboratoire J.-L. Lions, **Olivier.Pironneau@upmc.fr**

Groupe de travail du LJLL

Outline I

Complexité des architectures

- Multi-CPU et Multi-Processeurs
  - Plusieurs UAL en parallèle
  - Plusieurs CPU dans un même processeur
  - Plusieurs processeurs sur un même banc mémoire
  - Des grappes de cartes multiprocesseurs sur un réseau
- Hiérarchisation des mémoires
  - Mémoires périphériques
  - Mémoires principales RAM
  - Caches
  - Registres

Ordinateurs vectoriels

L'objectif est d'accélérer l'opération suivante

```
float x[N], y[N], z[N];
for (i = 0: i < N: i++)
z[i] = x[i] + y[i];
```

- p UA (vectoriel) ET la préemption des données (fetch) dans le cache.
- Automatisé si p et N sont petits.
- L'option -O3 dans la compilation ?

CBLAS

L'objectif est d'optimiser les opérations vectorielles bas niveau en utilisant une librairie adaptée à la machine `cblas_daxpy(..)` remplace y par $\alpha x + y$. Exemple: le gradient conjugué pour Ax=f

```
for(int iter=0;iter<n;iter++) {
    atimesx(A,x,grad);
    cblas_daxpy( n,-1,f,1,grad,1);        //  grad[i] -= f[i];
    double norm2min;
    double normg2 = cblas_ddot(n,grad,1,grad,1);
    if(!iter) norm2min = normg2*1.0e-8;
        if(normg2<norm2min)  break;
    double gamma = normg2/normg2old;
    cblas_dscal(n,gamma,h,1);  //  h[i] = gamma * h[i] - grad[i];
    cblas_daxpy( n, -1., grad,1, h, 1);  h[0]=0;
    double rho = cblas_ddot(n,h,1,grad,1); // rho = scal(h,grad);
    atimesx(A,h,grad);
    rho /= cblas_ddot(n,h,1,grad,1);      // rho /= scal(h,grad);
    cblas_daxpy( n, -rho, h,1, x, 1);     //  x[i] -= rho*h[i];
}
```

- Adapté du Fortran: blas1, blas2, blas3
- sur `edpblas.cpp` le cpu peut être divisé par 3!
- Remarque: il peut s'avérer difficile de trouver un cblas optimisé pour sa machine (ATLAS ?)

Super Ordinateurs et Ordinateurs en réseaux

- Ordinateurs vectoriels: SX9
- Ordinateurs massivements parallèles: Blue-Gene (IBM) etc.
- Cluster = un système par carte mère + une connectique rapide (myrinet, infiniband, ethernet gygabit)
- Ferme de stations: typiquement = plusieurs PC en réseau par ethernet
- Grid: Typiquement des machines sur la toile www. La grille EGEE permet d'accéder à 45 000 machines sur 240 sites

Les Ordinateurs disponibles

Vitesse en nb d'opérations flottantes par secondes (flops) (voir www.top500.org)

- machine intel centrino 2 a 2 ghz: 15 g flops
- et avec un GPU Nvidia Tesla (256 proc) : 0.5 tera flops
- Carte mère quadri-proc dual cores 3ghz: 80G flops
- Cluster 128 cartes bi-pro dual core 3 ghz: 2 tera flops
- La machine js21 du ccre: 5 tera flops
- Le SX8 vectoriel de l'Idris: 60 tera flops
- L'ibm blue-gene de l'Idris: 140 tera flops
- Le Road-runner de Los-Alamos: 1 peta flops

Les outils

- openMP
- MPI / mpich 2.0
- Globus et mpich-G
- upc-Berkeley
- CUDA

nb threads	omp gcc4.4	MPI	UPC	CUDA CPU	- GPU
1	0.9489		1.1388		
2	0.5647	0.5150	0.5805		
8		0.1547			
10		0.1316			
16		0.1412			
32				0.0207	0.1602

Exemple 1. Calcul d'une option en finance

- Le sousjacent S_t est modélisé par une EDO-S

$$dS_t = S_t(r dt + \sigma dW_t), \quad S(0) = S_0$$
$$\Rightarrow S_T = S_0 e^{(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}\mathcal{N}_{01}} \text{ si } r, \sigma \text{ constants}$$

- Le put est calculé par $P_0 = e^{-rT}\mathbf{E}(K - S_T)^+$
- La loi des grands nombres $\Rightarrow$ $P_0 \approx \frac{e^{-rT}}{M}(K - S_T^i)^+$
- On utilise des différences finies et on simule $dW_t = \sqrt{dt}\mathcal{N}_{01}$,

$$S_{m+1} = S_m + \delta t S_m(r\delta t + \sigma\sqrt{\delta t}\mathcal{N}_{01})$$

$\mathcal{N}_{01} = \sqrt{-2\log x}\cos(2\pi y)$ x,y aleatoires uniformes.

- Le calcul des S_T^i est "embarrassingly parallel".
- Voici le code C

edostoch.c(I)

```
#include <stdlib.h>
...
#include <time.h>
const int M=365; // nombre de pas de temps
const double two_pi =6.28318530718;

double gauss(){
    double x,y;
    x= (1.+rand())/(1.+RAND_MAX);
    y= (1.+rand())/(1.+RAND_MAX);
  return sqrt( -2 *log(x) )*cos(two_pi*y);
}

double EDOstoch(double S0, double dt, double sdt, double rdt){
    double S= S0;
    int i;  for(i=1;i<M;i++)
                  S= S*(1.+gauss()*sdt+rdt);
    return S;
}
```

edostoch.c (II)

```
int main(argc, argv) int argc; char *argv[];
{
    const double K=110, S0=100, T=1., r=0.03, sigma=0.2;
    const int k, kmax=20000; // nb de realisations
    double dt=T/M, sdt=sigma*sqrt(dt), rdt=r*dt, P0=0, t0=clock();
    srand(time(NULL));

    for(k=0; k<kmax;k++){
        double Sa= EDOstoch(S0, dt, sdt, rdt);
        if(K>Sa) P0 += K-Sa;
    }
    t0=(clock()-t0)/CLOCKS_PER_SEC;
    printf("P_0 = %f CPUtime=%f \n",P0*exp(-r*T)/kmax, t0);
    return 0;
}
```

Outline I

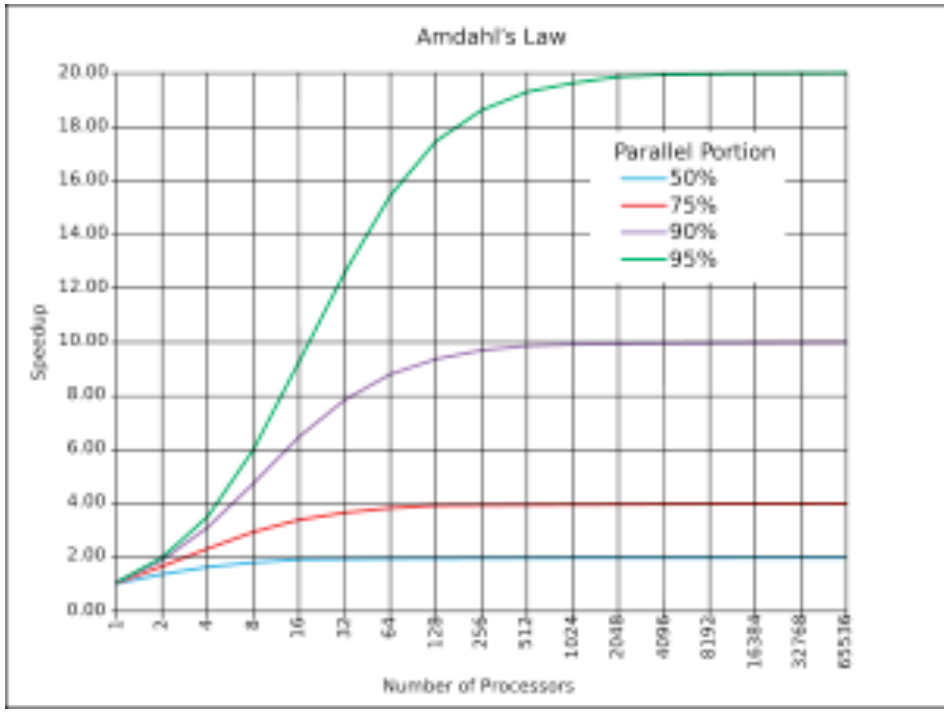
Le code edostoch.cpp en openMP

```
int main(){
   const double K=110, S0=100, S[P];    srand(time(NULL));
   for(int q=0; q<P;q++)  S[q]= S0;
   double time0=omp_get_wtime();
#pragma omp parallel  for num_threads(P)
   for(int q=0; q<P;q++)   {
      srand(q+time(NULL));
      for(int k=0; k<kmax/P;k++){
         double Sa= EDOstoch(S0);
         if(K>Sa) S[q] += K-Sa;
      }
   }
   time0-=omp_get_wtime();
   double PT = S[0];
   for(int q=1; q<P;q++)  PT += S[q];
   printf("P_T = %f CPUtime=%f\n",PT*exp(-r*T)/kmax, -time0);
   return 0;}
```

Performance avec gcc 4.4 sur un macbook pro:

`CPUtime(1)=0.948938 CPUtime(2)=0.565159`

Loi de Amdhal



Loi de Amdhal: le speed-up est limité par la partie séquentiel du programme. Le speed-up est S/[(1-p)S+pS/N] ou S est le temps calcul sequentiel,p la proportion parallélisée et N le nb de processeurs.

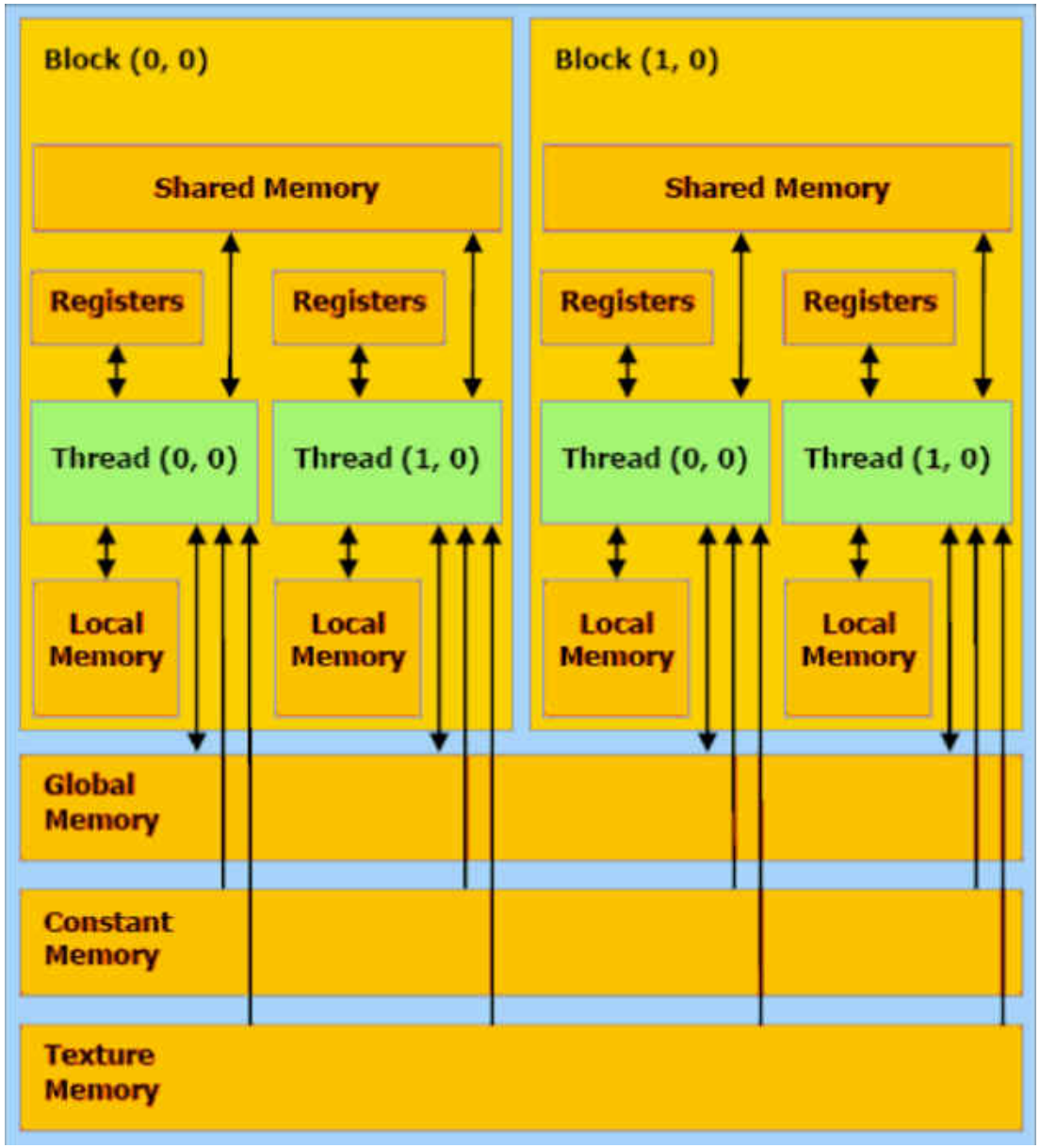
Exercice: Obtenir le meilleur speed-up avec openMP sur edostoch.c

Outline I

Graphic Processor Units

- Le marché du jeu video à induit une concurrence féroce entre les 2 grands constructeurs ATI et Nvidia.
- Le besoin de réalisme a obligé les concepteurs de jeux à revenir vers les équations fondamentales de la physique pour la simulation, en particulier pour l'eau et la fumée.
- Vers 2006 les unités de calcul élémentaires sont devenues capables de calculer en virgule flottante: le GPGPU (general purpose graphic processor unit).
- Des chercheurs comme Pat Hanrahan (Stanford) ont développé des langages dédiés comme *brook*, puis CUDA et OpenCL.
- Intel travaille sur *Larrabee* un processeurs sur le principe des GPGPU: 32 CPU avec des mémoires hiérarchiques et des communications rapides.

Le Modèle de mémoire Nvidia (I)

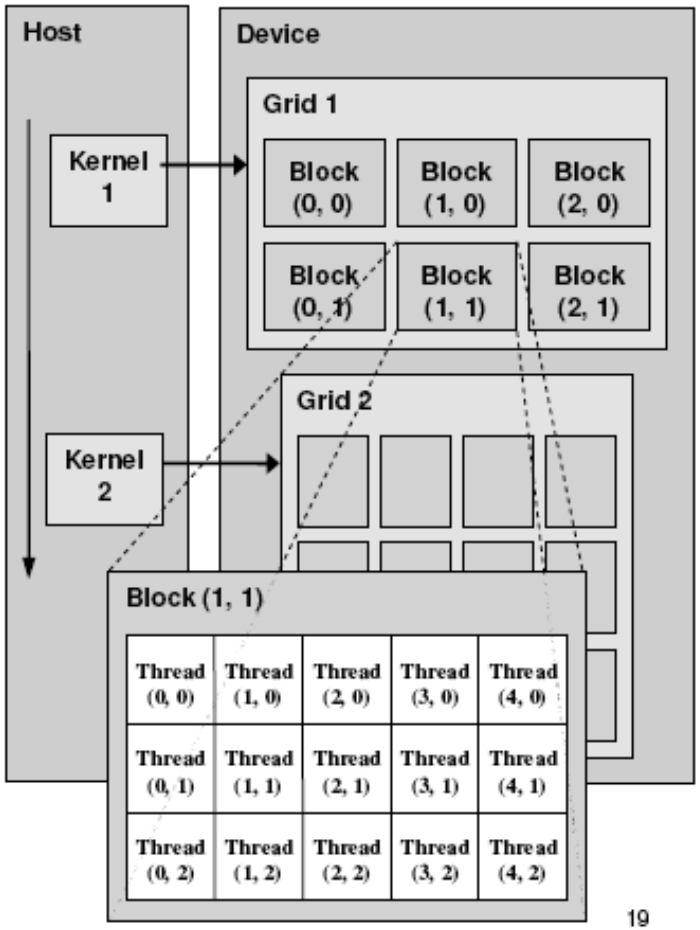


Le Modèle de mémoires de Nvidia (III)

CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



19

Programmation en CUDA

Nous allons ecrire un priceur d'option put basé sur la formule

$$S_T^n = S_0 e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}N_{01}^N}, \quad P_T = \frac{e^{-rT}}{N}\sum_{n=1}^{N}(K - S_T^n)^+$$

Cette formule vient d'une solution analytique de l'EDS de Black-Scholes pour S_t lorsque r et σ sont constants. Nous allons utiliser la formule de Cox-Muller pour générer les réalisations N^n de la variable aléatoire gaussienne N :

$$N_{01} = \sqrt{-2\log(x)}\cos(2\pi y), \quad x, y \text{ aleatoires uniformes sur } (0,1)$$

- x^n, y^n sont générées par `rand()` dans le CPU et
- envoyées au GPU en recopiant deux tableaux A1,A2 en RAM dans B1,B2, memoires de la carte graphique.
- Les 2 formules ci-dessous sont évaluées dans le GPU pour chaque x^n, y^n de $B1, B2$ et
- le resultat est stocké dans B2 et renvoyé par recopie dans A3.

Edition-compilation-exécution

Le plus simple est d'utiliser un Macbook avec une carte Nvidia. On peut utiliser Xcode et Eclipse ou une fenetre terminal.

- Le site de Nvidia permet d'installer un binary tout pret dans `/usr/local/bin`
- Pour pointer sur le compilateur il faut faire
```
export PATH=/usr/local/cuda/bin:$PATH
export DYLD_LIBRARY_PATH
          =/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```
- Pour compiler il faut `nvcc BSCuda.cu`
  et en mode emul: `nvcc -deviceemu BSCuda.cu`
- Pour lancer l'exécutable il faut `./a.out`
- **Function Type Qualifiers**
  - `__device__` appelée par le GPU et exécutée sur le GPU
  - `__global__` appelée par l'hôte et exécutée par le GPU
  - `__host__` appelée par l'hôte et exécutée par l'hôte
- Variables qual.: `__device__`, `__shared__`, `__constant__`

```
#include <math.h>
#define PI 3.14159265358979323846264338327950288f
const int NbBlocs = 20000, NbThreads = 500, N  = NbBlocs*NbThreads;
const float  K = 110, S0=100, r=0.02, sig=0.3, T=1.0,R = (r-sig*sig/2)*T;

__host__ __device__ void BS(float *x, float *y){
   float z = sqrtf(-2.0f * logf(*x)) * cosf(2.0f * PI * (*y));
   *y= S0*expf(R+sig*sqrtf(T)*z);
}
__global__ void BSgpu(float *a1, float *a2, int l) {
   int i = blockDim.x*blockIdx.x + threadIdx.x;
   if ( i < l ) BS(a1+i, a2+i);
}
void BScpu(float *a1, float *a2, int l) {
   for ( int i = 0 ; i < l ; ++i )  BS(a1+i, a2+i);
}

int main() {
   const int taille = N*sizeof(float);
   float *A1 = (float *) malloc(taille); // allocate A in CPU RAM
   float *A2 = (float *) malloc(taille);
   float *A3 = (float *) malloc(taille);
   float *B1, *B2;                       // will be allocated in GPU RAM
   srandom(time(NULL));
    for ( int n = 0 ; n < N ; ++n ){  // fill vector A with random nb
      A1[n] = (rand() + 0.5f)/(RAND_MAX + 1.0f);
      A2[n] = (rand() + 0.5f)/(RAND_MAX + 1.0f);
   }
   cudaMalloc( (void **) &B1, taille);           // allocate B1 in GPU RAM
   cudaMemcpy(B1, A1, taille, cudaMemcpyHostToDevice); // transfer A1 into B1
   cudaMalloc( (void **) &B2, taille);
   cudaMemcpy(B2, A2, taille, cudaMemcpyHostToDevice);

   BSgpu<<<NbBlocs, NbThreads>>>(B1,B2,N);

   cudaMemcpy(A3, B2, taille, cudaMemcpyDeviceToHost); // transfer results in A3

   BScpu(A1,A2,N);

   float put=0, err = 0;
   for ( int n = 0 ; n < N ; ++n ){
      err += fabs(A3[n] - A2[n]);
      put += fmax(K-A2[n],0.0f);
   }
   return 0;
```

Calcul sur les machines de Chevaleret et autres

- Pour ceux qui ont un compte au LJLL Khashayar Dadras a mis une PC-Ubuntu avec une Tesla sur le réseau. C'est la machine `nice.ann.jussieu.fr`
- Pour ceux qui ont un compte au LPMA Jacques Portes a mis sur le réseau la machine `Tesla.proba.jussieu.fr`.
- Pour les applications *computer intensive* la machine Tera du CCRT (CEA-civil) aura une extension avec 48 cartes a base de Tesla. En attendant il y a une petite config au CINES accessible par www.dari.fr.

Equation de la chaleur et de Black-Scholes

$$\partial_t u - \partial_x(\kappa \partial_x u) = f, \quad u(0,t) = u(L,t) = 0, \quad u(x,0) = u^0(x) \ \ \forall x, t \in (0,L)$$

Remarque: L'EDP de Black-Scholes est du même type.

Formulation variationnelle et differences finies en temps

$$\int_0^L \frac{u^{m+1} - u^m}{\delta t} w(x)\mathrm{d}x + \int_0^L \kappa \partial_x u^{m+1} \partial_x w = \int_0^L f w \ \ \forall w \in V := H_0^1(0,L)$$

Discretisation en espace par éléments finis de degrés 1: on remplace V par V_h, l'espace des fonctions continues affines par morceaux sur $[0,L] = \cup_i [x_i, x_{i+1}]$ avec $x_i = ih$,i=0..I-1, tel que $Ih = L$. On obtient un système lineaire a chaque itération pour $U^{m+1} \in \mathcal{R}^N$:

$$B(U^{m+1} - U^m) + AU^{m+1} = F \in \mathcal{R}^N,$$

$$\text{avec } B_{ij} = \frac{1}{\delta t} \int_0^L w^i w^j \mathrm{d}x, \ A_{ij} = \int_0^L \kappa \nabla w^i \nabla w^j \mathrm{d}x$$

où w^i est la fonction de V_h qui vaut δ_{ij} en x_j.

Equation de la chaleur: discretisation

Il est facile de voir que A et B sont tridiagonales avec

$$B_{ii} = \frac{2h}{\delta t},\ B_{i,i-1} = B_{i,i+1} = \frac{h}{\delta t},\quad A_{ii} = \frac{2\kappa}{h},\ A_{i,i-1} = A_{i,i+1} = -\frac{\kappa}{h}$$

A priori le système tridiagonal pour U^{m+1} est résolu par factorisation de Gauss . Mais la parallélisation de la méthode du gradient conjugé est beaucoup plus simple. On rappelle l'algorithme pour résoudre $Ax = b$:

1. $g^k = Ax^k - b$
2. $h^k = -g^k + \gamma h^{k-1}$
3. $x^{k+1} = x^k + \rho h^k$

avec $\gamma = \|h^k\|/\|h^{k-1}\|$ et $\rho = -g^k \cdot h^k / h^k \cdot Ah^k$

Outline I

La fonction centrale de `vanila.cpp`

```
void Option::calc(bool refact )
{ const int im = xmin/dx;
  for(int i=0;i<nX;i++) u[i] = phi(K,i*dx);
  for(int j=0;j<nT;j++) {     // time loop
        for(int i=1;i<nX-1;i++)  // rhs of PDE
          uold[i] = u[i] + dt*r*(i+im)*(u[i+1]-u[i-1])/2; // explicit part

  u[nX-1]=0;     // B.C. of PDE
  if(im>0) u[0]=0; else u[0] = K*exp(-r*(j+0.5)*dt);
  uold[1] += u[0]*sigma[j][1]*sigma[j][1]*dt/2;

  if(j==0 || refact){
       for(int i=1;i<nX-1;i++) { // build matrix
            double aux=(i+im)*sigma[j][i]*(i+im)*sigma[j][i]*dt/2;
            bm[i] = (1+ r*dt + 2*aux);
            am[i] = -aux;
            cm[i] = -aux;
          }
       factLU();
  }
  for(int i=1;i<nX-1;i++) u[i]=uold[i];
  solveLU(u);
   }
```

Rajouter `#pragma openmp parallel for num_threads(nthreads)` avant les boucles en nX.

L'appel de la fonction centrale dans le main()

```
int main(){
   const double r=0.03, sig=0.3, K=110, T=1, Sm=0, SM=300;// no barrier
//  For  up-barrier make SM smaller than 3*K, for low barrier use Sm>0
   Option p(150,300,SM,sig,r,K,T,Sm);// nT, nX...

   p.calc(0);

  ofstream fout("put.txt"); // for display with gnuplot
  cout.setf(ios::fixed);
  cout<<"S_0 \t\t Put by FEM\t Put by BS \t Error"<<endl;
  for(int i=0;i<p.nX;i++){
   if(i%15==0){ cout<<setw(5)<<Sm+i*(SM-Sm)/p.nX<<"\t"<<setw(5)<<p.u[i];
        if(Sm==0) { double bsput = BSPut(i*SM/p.nX, T, r, K, sig);
          cout<<"\t"<<setw(5)<<bsput<<"\t" <<setw(5)<<bsput-p.u[i];
        }
        cout <<endl;
   }                        // display with gnuplot using: plot"put.txt"
   fout<<Sm+i*(SM-Sm)/p.nX<<"\t"<<p.u[i]<<endl;
  }
  return 0;
}
```

Produit Matrice Vector avec `cblas_sgmv`

```
  #else  // blas simule':
    int CblasRowMajor,CblasNoTrans;
    int cblas_sgbmv (int row,int trans,int sizc,int sizr,int dum1,int dum2,
           real b,real* A,int bandA,real* u,int dum3,real c,real* w,int dum4){
     int i;
     if(c)
       for (i = 1; i < sizc-1; i++)
          w[i] = b*(A[i]*u[i] + A[i+1]*u[i+1]+A[i-1]*u[i-1]) + c*w[i];
     else
       for (i = 1; i < sizc-1; i++)
          w[i] = b*(A[i]*u[i] + A[i+1]*u[i+1]+A[i-1]*u[i-1]) ;
     return 0;
     }
  #endif
#endif
```

La fonction `cblas_sgmv` fait la même chose que ce qui lui suit.

$u \longrightarrow w := Au$, A tri-diagonal, avec cublas

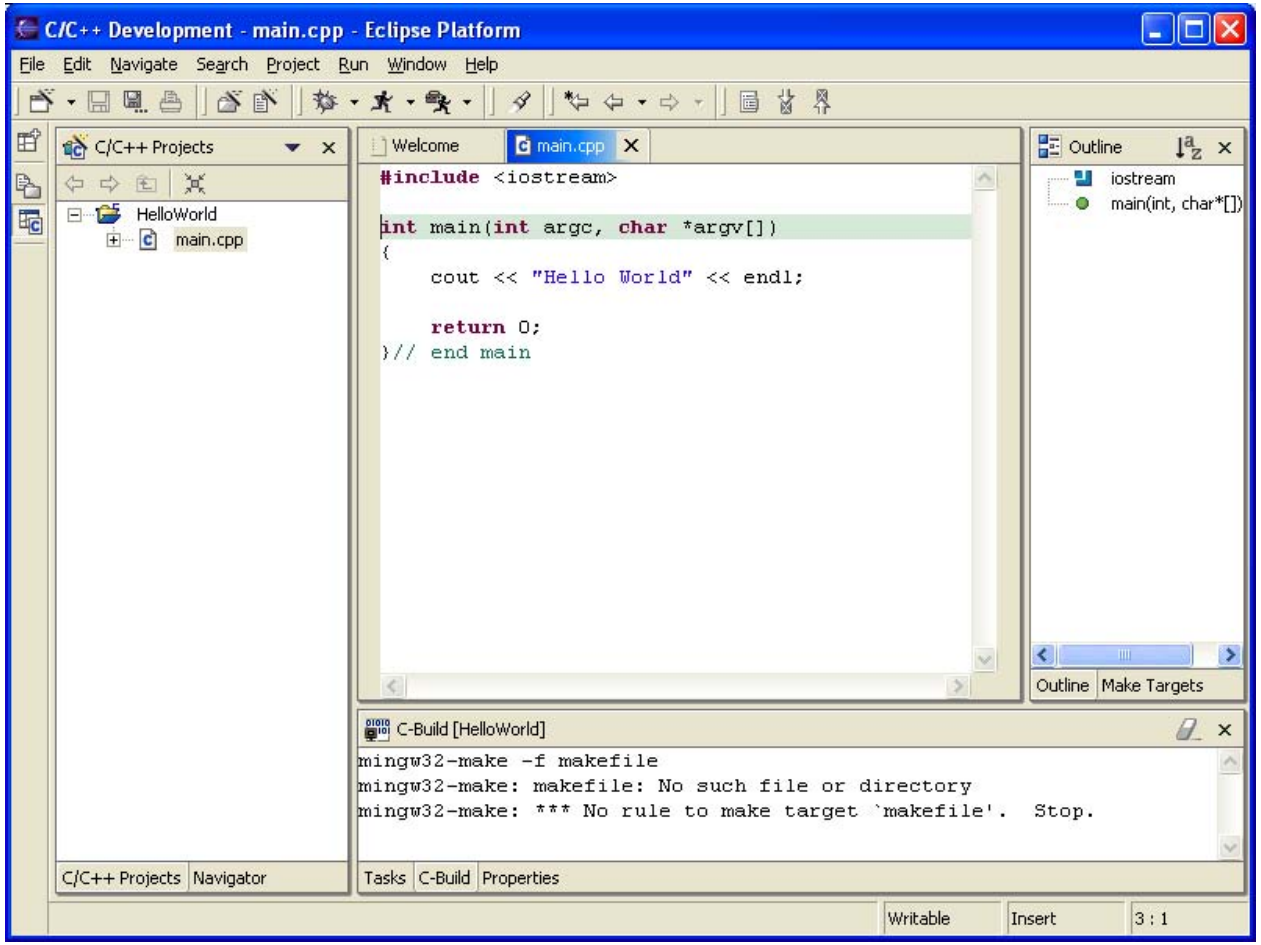
```
#include "cublas.h"
/*    export PATH=/usr/local/cuda/bin:$PATH
      export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
      compile with nvcc -lcublas testcu.c
*/
#define M 32000
    int main (void){
    int i;
    float *devPtrA, *ptrw=0, *ptru=0, *A = 0, *w=0, *u=0;
    A = (float *)calloc (M *3+ 3 , sizeof (*A));
    w =     (float*)calloc(M,sizeof(float));
    u =     (float*)calloc(M,sizeof(float));
    for (i = 0; i < 3*M+3; i++)  A[i] = i;
    w[5] = 1;
    cublasStatus stat;
    cublasInit();
    stat = cublasAlloc (M, sizeof(*w), (void**)&ptrw);
    cublasSetVector(M, sizeof(*w), w, 1, ptrw,1);
    stat = cublasAlloc (M, sizeof(*u), (void**)&ptru);
    cublasSetVector(M, sizeof(*u), u, 1, ptru,1);
    stat = cublasAlloc (M*3+3, sizeof(*A), (void**)&devPtrA);
    cublasSetVector(3*M+2, sizeof(*A), A, 1, devPtrA,1);
cublasSgbmv ('n', M, M, 1, 1, 1.0, devPtrA, 3, ptrw, 1, 0.0,ptru, 1);
cublasGetVector(M, sizeof(float), ptru, 1, u,1);
    cublasFree (ptrw);
    cublasFree (devPtrA);
    cublasShutdown();
    return 0;
}
```

Gradient Conjugué cublas

```
void cublas_gradconj(Vector A, Vector h, Vector grad, Vector f, Vector x) {
    float normg2old = 1e30;
    float norm2min,normg2,gamma;
    int iter,n=nX;
    for(iter=0;iter<n;iter++)
    {
        cublasSgbmv ('n', n, n, 1, 1, 1.0, A, 3, x, 1, 0.0, grad, 1);
        cublasScopy(1,f,1,grad,1);  // so that grad[0]=0;
        cublasSaxpy( n,-1,f,1,grad,1);
        normg2 = cublasSdot(n,grad,1,grad,1);
        if(!iter) norm2min = normg2*1.0e-8;
        if(normg2<norm2min)  break;
        gamma = normg2/normg2old;
        normg2old = normg2;
        cublasSscal(n,gamma,h,1);
        cublasSaxpy( n, -1., grad,1, h, 1);
        float rho = cublasSdot(n,h,1,grad,1);
        cublasSgbmv ('n',n, n, 1, 1,1.0,A, 3,h, 1, 0.0,grad, 1);
        rho /= cublasSdot(n,h,1,grad,1);
        cublasSaxpy( n, -rho, h,1, x, 1);
    }
}
```

Malheureusement les perfs ne sont pas au rendez-vous, essentiellement parce que ce qui est en dehors de cublas se fait dans le CPU et implique des communications non gérées.

L'environnement de travail Eclipse



Multi plateforme et gratuit (suppose que gcc et un java installés)
Eclipse est configurable pour openMP, MPI, UPC, CUDA via PTP

Outline I

Présentation de UPC

Proposé en 1999, UPC est dévelopé par un consortium dont Berkeley fait partie. Berkeley Unified Parallel C compiler tourne sur les principaux environnements. La compilation et l'exécution se font par:

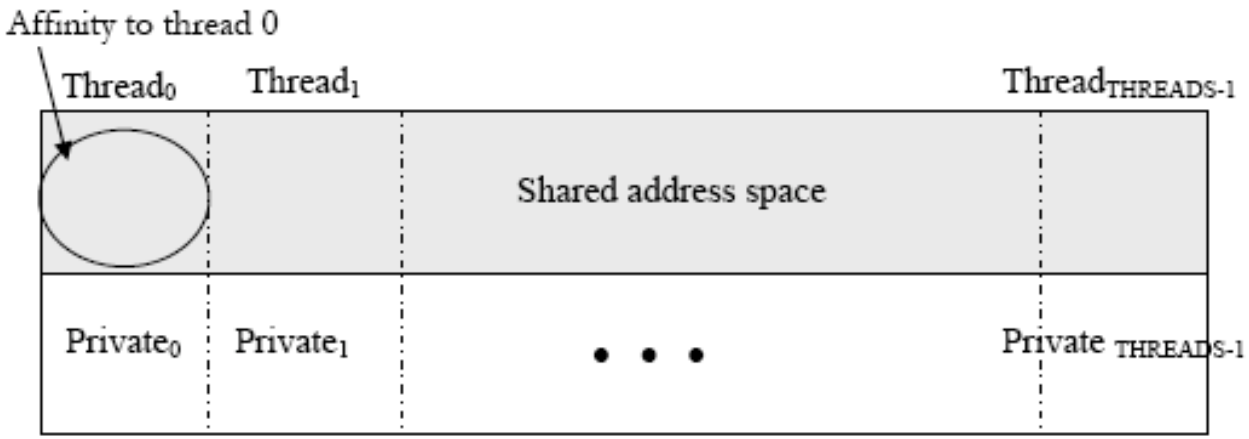
```
upcc -pthreads hello.upc -o hello
upcrun -n 2 ./hello
```

Il reprend des idées de MPI mais simplifie enormément les communications en introduisant la notion de shared variable.

Installation

- Sur Mac, il existe un binary qu'il suffit d'installer en suivant la doc
- Sur Ubuntu il faut descentre le .tar du runtine et l'installer avec `./configure` et `make install`. La compilation se fait sur http://upc.lbl.gov.
- Sur XP/Vista un tar d'une installation complete de cygwin+MPI+upc est proposée !
- voir http://upc.lbl.gov

Organisation mémoire



Le programme est recopié sur chaque processeur, chaque variable est donc en THREADS exemplaires sauf si elle est déclaré `shared`; dans ce ca elle est par defaut sur la mémoire du process 0 et distribué si déclaré comme telle:

```
#define N 1000
int i;
shared double x;
shared [THREADS] double a[N];
```

Chaque proc accède a toute variable shared et connait son affinity

Exemple: Produit matrice-vecteur

```
#include<upc_relaxed.h>
#define N 100*THREADS
    shared [N] double A[N][N];
    shared double b[N], x[N];

void main(){
  int i,j;
  upc_forall(i=0;i<N;i++;i)
  for(j=0;j<N;j++)
    b[i]+=A[i][j]*x[j] ;
}
```

Le `upc_forall` est un “parallel for” où le dernier argument indique qui fera l’opération. Ici l’affinité de *i* détermine le proc: comme *i* est local, c’est lorsque le *i* de la boucle est egal au i local. On aurait pu écrire:

```
for(i=0;i<N;i++)
   if(MYTHREAD==(i%THREADS))  b[i]+=A[i][j]*x[j] ;
```

Attention: ca se complique si N n’est pas un multiple de THREADS.

Exemple 1: edostch.upc

```
#include <upc_relaxed.h>
#include <sys/time.h>
// upcc -pthreads edostoch.upc
// upcrun -n 2 ./a.out
const double two_pi =6.28318530718;
shared double PT;
shared double runtimes[THREADS];

int main(){
    struct timeval ts_st, ts_end;
...
    gettimeofday( &ts_st, NULL );

    if(MYTHREAD==0) PT=0;
    dt=T/M; sdt =sigma*sqrt(dt); rdt = r*dt;

    upc_forall(i=0;i<THREADS;i++;i) {
        srand(i+time(NULL));
        for(k=0; k<kmax/THREADS;k++){
            double Sa= EDOstoch(S0, dt, sdt, rdt,M);
            if(K>Sa) S += K-Sa;
          } PT+=S;
    }
  gettimeofday( &ts_end, NULL );
  upc_barrier;
  runtimes[MYTHREAD] = ts_end.tv_sec-ts_st.tv_sec
      + (ts_end.tv_usec - ts_st.tv_usec) / 1000000.0;
...   return 0;
}
```