Параллелизм в итерационных методах решения систем линейных уравнений

(Занятие 6)

Игорь Николаевич Коньшин

МГУ - 05.07.2012

Предобусловливание

$$A\,x = b$$

- Приближенное треугольное разложение ILU:

$$A \sim L\,U$$

- Построение предобусловливателя $H = U'\,L'$ для предобусловливания исходной системы:

$$A\,x = b \quad \rightarrow \quad (U'\,L'\,A)\,x = U'\,L'\,b \quad \rightarrow \quad A''\,x = b''$$

$$A'' = \; U'\,L'\,A \sim I$$

- *Решение новой системы с матрицей близкой к единичной – мечта любой итерационной схемы*

Неполные и приближенные треугольные разложения

Разложение на треугольные сомножители: $A \approx \tilde{L}\tilde{U}$

$$a_{ij} := a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$$

Неполные разложения (разложения "по позициям", или искусственное априорное ограничение структуры L и U):
(*) ILU(k) по структуре A^{k+1}
(*) блочнодиагональные L и U

Приближенные разложения (разложение "по значениям" элементов, разложение 1-го порядка точности):
(*) τ - "отсечение" малых элементов
(*) δ - предварительный диагональный сдвиг: $A + \delta I$
(*) pivmin - минимальный ведущий элемент
(*) ncol - максимальное число элементов в строке

Приближенные разложения (2-го порядка точности) ILU2:
(*) τ - "отсечение" малых элементов
(*) τ^2 - точность вычислений
(*) $\delta = \tau^2$ - предварительный диагональный сдвиг: $A + \delta I$

Применение специальных упорядочиваний для минимизации возникающих элементов в L и U (минимизация ширины профиля, RCM)

Итерационные методы

Симметричные линейные системы:

- Метод сопряженных градиентов

  CG (Conjugate Gradient)

Несимметричные линейные системы:

- Стабилизированный метод бисопряженных градиентов, BiCGStab
- Метод минимальных невязок

  GMRES (Generalized Minimal RESidual)
- SOFGMRES...

Метод сопряженных градиентов

Для решения системы линейных алгебраических уравнений

$$Ax = b,$$

где

A – разреженная с.п.о. $(n \times n)$-матрица,
b – правая часть,
x – искомое решение,

рассмотрим МСГ:

$$r_0 = b - Ax_0, \quad p_0 = Hr_0, \qquad i = 0, 1, \ldots,$$

$$\alpha_i = \frac{r_i^T H r_i}{p_i^T A p_i}, \quad x_{i+1} = x_i + p_i \alpha_i, \quad r_{i+1} = r_i - A p_i \alpha_i,$$

$$\beta_i = \frac{r_{i+1}^T H r_{i+1}}{r_i^T H r_i}, \quad p_{i+1} = H r_{i+1} + p_i \beta_i,$$

где

$H \sim A^{-1}$ – предобусловливатель для матрицы A,
x_0 – начальное приближение к решению,
x_i – решение на i-ой итерации МСГ,
$r_i = b - Ax_i$ – невязка на i-ой итерации.

Прямые и итерационные методы решения

Характерное время решения СЛАУ

прямым и итерационным методом:

[<– – – – – – – – – – – – –прямой метод– – – – – – – – – – – – –>]

[<– – – – – – – – – – – – –Iter– – – – – – – – – – – – –>] CG

[<Prec><– – – – – – – –Iter– – – – – – – –>] PCG+IC(0)

[<– –Prec– –><– – – – –Iter– – – – –>] PCG+IC(2)

[<– – –Prec– – –><– – –Iter– – –>] PCG+IC2

Примеры из практики

Презентация BIIC + IC2 + PCG

Эффективность // приложений

- Оценка эффективности
- Эффективный // алгоритм
- Эффективная // программа

Оценка ускорения

p - количество используемых процессоров

T(p) - время решения задачи на p процессорах

S - ускорение, S=T(1)/T(p)

E - эффективность, E=S/p

La - общее количество арифметических операций алгоритма

Lc - общая длина всех обменов данными для алгоритма

ta - среднее время выполнения одной арифметической операции

tc - среднее время выполнения обмена одним числом

Ta = La ta - время затраченное на арифметику (вычисления)

Tc = Lc tc - время затраченное на коммуникации (обмены)

tau = tc / ta - общая характеристика параллельного компьютера

L = La / Lc - общая характеристика параллельности алгоритма

$$S = S(p) = T(1) / T(p) = Ta / (Ta/p + Tc/p) = p\,Ta / (Ta + Tc) = p / (1 + Tc/Ta) =$$
$$= p / (1 + (Lc\,tc) / (La\,ta)) = p / (1 + tau / L)$$

Оценка // эффективности

Ускорение:

$$S = p / (1 + tau / L)$$

Эффективность:

$$E = S / p = 1 / (1 + tau / L)$$

$$E \sim 1 - tau / L, \qquad tau / L \ll 1$$

Потеря эффективности: tau / L

tau - мера //-ти компьютера

L - мера //-ти алгоритма

«Идеальная» // программа

- сами вычисления реализованы эффективно

  (т.е. программа хорошо работает и на одном процессоре)
- обмены выполняются асинхронно на фоне вычислений

  (т.е. не тратится время на ожидание завершения обменов)
- вычислительная нагрузка на процессоры хорошо сбалансирована

  (т. е. нет простоя процессоров)

Перспективные // архитектуры

- Многоядерные процессоры
  - Intel
  - AMD
  - IBM
- Графические ускорители
  - Nvidia

Intel Core i7

Memory Bus Controller							
L3 cache							
L2 cache		L2 cache		L2 cache		L2 cache	
L1-I	L1-D	L1-I	L1-D	L1-I	L1-D	L1-I	L1-D
P0		P1		P2		P3	

NVIDIA Tesla

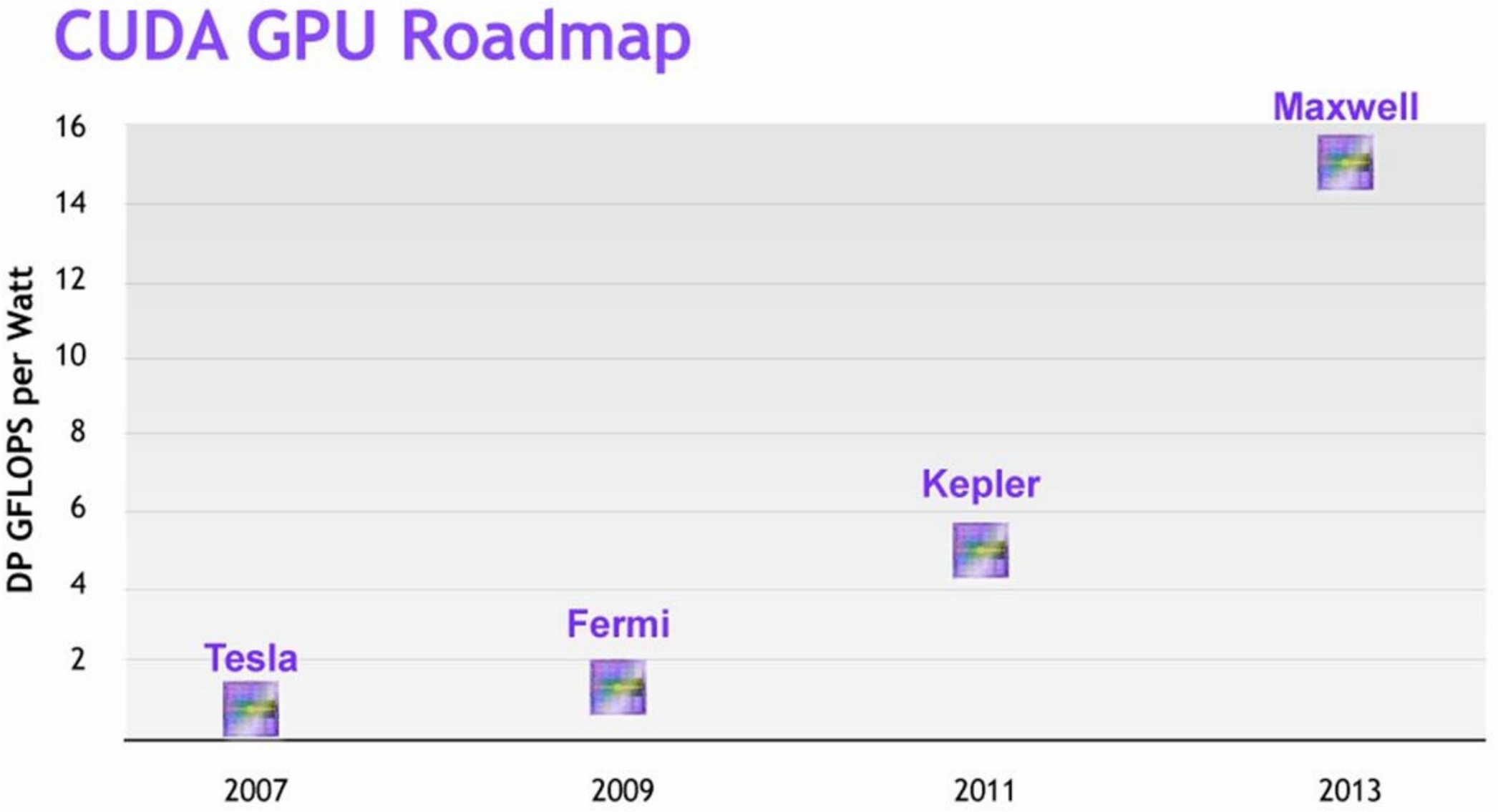
GPU: особенности

- Очень много ядер (до 500)
- Много функциональных устройств
- Иерархия памяти
- Энергоэффективность
- Обгоняет CPU по производительности

- (±) Специальная среда программирования Nvidia CUDA

- (-) Ограничения по функциональности

GPU: перспективы



Пример программы на CUDA (блочный DOT: xy = x * y)

```
void BdotD(int _n, int _m, DoubleV *_x, DoubleV *_y, DoubleV *_xy)
{
  DoubleV *d_w;
  int threadsPerBlock = (128 / (_m * _m)) * _m * _m; // 128 is the number of threads per block
  int blocksPerGrid = 64;                            // number of multiprocessors times some coefficient

  int nred = blocksPerGrid * _m * _m;
  cuSafeCall(cudaMalloc((void**) &d_w, sizeof(DoubleV)*nred));

  if (_m == 4)
    BdotD4_kernel<<<blocksPerGrid, threadsPerBlock>>>(_n, _m, _x, _y, d_w);
  else
    BdotD0_kernel<<<blocksPerGrid, threadsPerBlock>>>(_n, _m, _x, _y, d_w);

  cuSafeCall(cudaDeviceSynchronize());

  threadsPerBlock = 64;
  blocksPerGrid   = _m * _m;
  BdotD_64_kernel_reduce<<<blocksPerGrid, threadsPerBlock>>>(_m, nred, d_w, _xy);

  cuSafeCall(cudaFree(d_w));
}
```

Программа-ядро для GPU: dot

```
__global__ void BdotD4_kernel(int _n, int _m, double *_x, double *_y, double *_res)
{
  const int m    = 4;
  const int mm   = m * m;
  const int dd   = mm * 8; // i.e. 8 semi-warps per block, =128;
  const int step = gridDim.x  * blockDim.x;
  const int id   = blockIdx.x * blockDim.x + threadIdx.x;
  const int is   = threadIdx.x;
  const int imm  = is % mm;
  const int i    = imm % m;
  const int j    = imm / m;
  const int kx   = is - imm + i;
  const int ky   = is - imm + j;
  const int kend = ((_n * m) / step) * step;
  int k;
  __shared__ double sx[dd], sy[dd], ss[dd];

  // Main part of the loop
  DoubleV sum = (DoubleV)0.0;
  for (k=id; k<kend; k+=step) {
    sx[is] = _x[k];
    sy[is] = _y[k];
    __syncthreads();
    sum += sx[kx]*sy[ky] + sx[kx+m]*sy[ky+m] + sx[kx+m*2]*sy[ky+m*2] + sx[kx+m*3]*sy[ky+m*3];
    __syncthreads();
  }
```

Программа-ядро для GPU: dot
(продолжение...)

```
   // Tail of the loop
   sx[is] = 0.0;
   sy[is] = 0.0;
   k = kend + id;
   if (k < _n * m) {
      sx[is] = _x[k];
      sy[is] = _y[k];
   }
   __syncthreads();
   sum += sx[kx] * sy[ky] + sx[kx+m] * sy[ky+m] + sx[kx+m*2] * sy[ky+m*2] + sx[kx+m*3] *
sy[ky+m*3];

   ss[is] = sum;
   __syncthreads();

   // Local sum inside each block
   if (is < mm) {
      sum = 0.0;
      for (k=is; k<blockDim.x; k+=mm)
         sum += ss[k];
      _res[blockIdx.x*mm+is] = sum;
   }
}
```

Программа-ядро для GPU: reduce

```
__global__ void BdotD_64_kernel_reduce(int _m, int _nred, double *_w, double *_xy)
{
   const int mm = _m * _m;
   const int dd = 64;
   const int tid = threadIdx.x;
   const int bid = blockIdx.x;
   double sum;
   __shared__ double ss[dd];
   ss[tid] = sum = _w[bid + tid * mm];
   __syncthreads();
   if (tid < 32) {
      ss[tid] = sum = sum + ss[tid + 32]; __syncthreads();
      ss[tid] = sum = sum + ss[tid + 16]; __syncthreads();
      ss[tid] = sum = sum + ss[tid +  8]; __syncthreads();
      ss[tid] = sum = sum + ss[tid +  4]; __syncthreads();
      ss[tid] = sum = sum + ss[tid +  2]; __syncthreads();
      ss[tid] = sum = sum + ss[tid +  1]; __syncthreads();
      if (tid == 0)
         _xy[bid] = sum;
   } else {
      __syncthreads(); __syncthreads(); __syncthreads(); __syncthreads(); __syncthreads(); __syncthreads();
   }
}
```

Схема параллелизма

- 3-х уровневый параллелизм:

  MPI + TBB + CUDA

  (проц-ры) (нити CPU) (нити GPU)

- Иерархия зависимостей:

  MPI --> TBB --> CUDA