Multi Core Computing

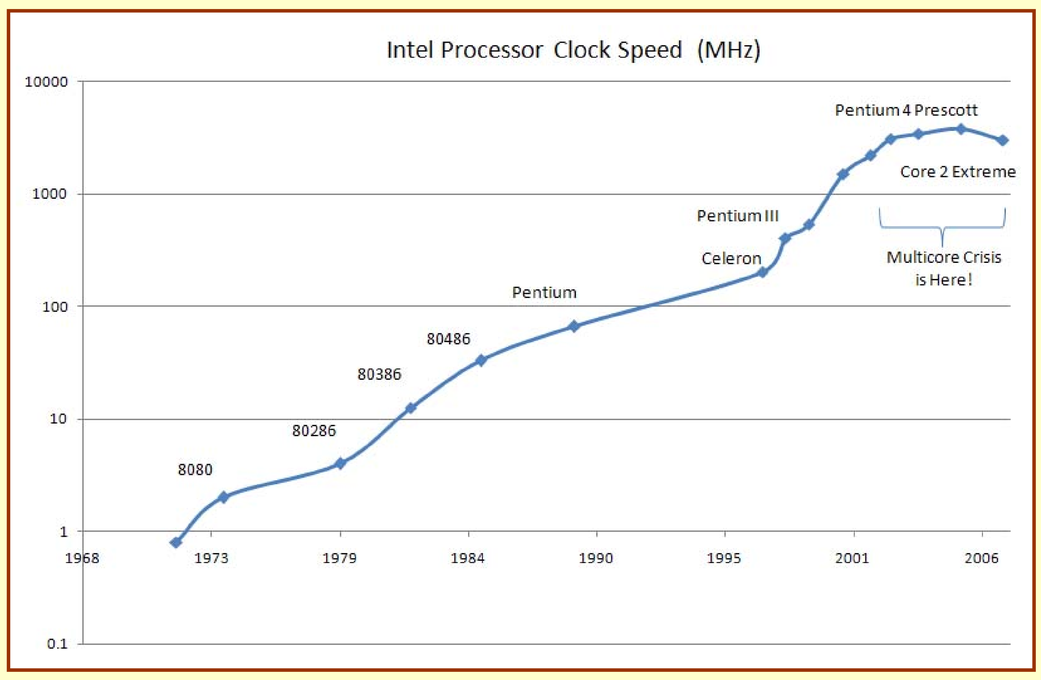
Richard Ansorge

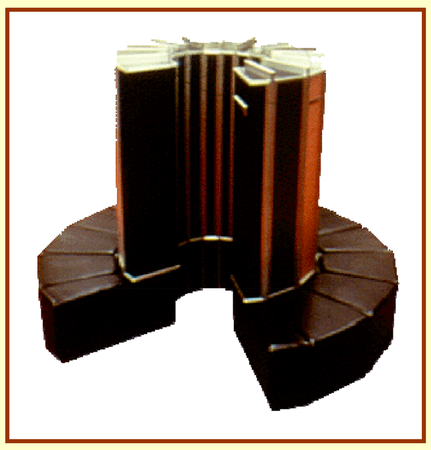
January 2010

Moore's Law?

- Exponential growth in computing power since early 1960's.
- Single processor power peaked in 1990's
- Many calculations limited by memory bandwidth not CPU
- PC clusters used since mid 1990's
- Multicore PC chips now entry level
- Graphics cards have 100's of cores

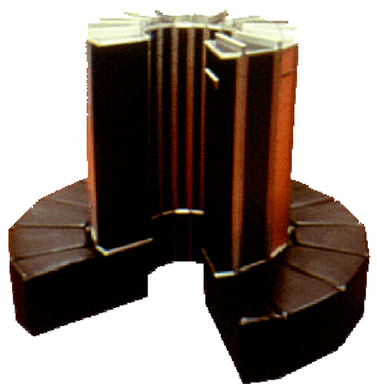
Moore's Law





Cray 1

In 1975 the 80 MHz Cray-1 was announced. Excitement was so high that a bidding war for the first machine broke out between Lawrence Livermore National Laboratory and Los Alamos National Laboratory, the latter eventually winning and receiving serial number 001 in 1976 for a six-month trial. The National Center for Atmospheric Research (NCAR) was Cray Research's first official customer in July 1977, paying US$8.86 million ($7.9 million plus $1 million for the disks). The NCAR machine was decommissioned in January 1979. The company expected to sell perhaps a dozen of the machines, but over eighty Cray-1s of all types were sold, priced from $5M to $8M. The machine made Cray a celebrity and the company a success, lasting until the supercomputer crash in the early 1990s.



Peak 0.25 Gflops

5.5 Tons wt

110-250 kW

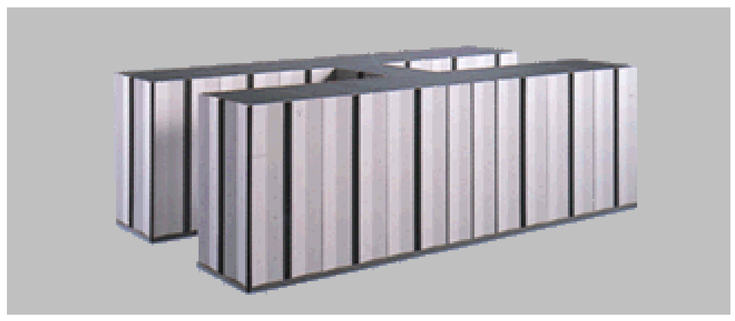
Liq Freon cooled

Distributed Computing

- If lucky can just run same code on each processing core; e.g. event simulation.
- But be careful to use 64 bit random number generator and ensure different sequences.
- Otherwise cores share data:
  - shared memory machine
  - local memories + interconnect
  - latency and bandwidth constraints

SR2201 – Cambridge HPCF 1996

- Very expensive up to 1024 cores
- Fast interconnect
- data streaming to beat memory bandwidth
- Pi to 50 10^9 places

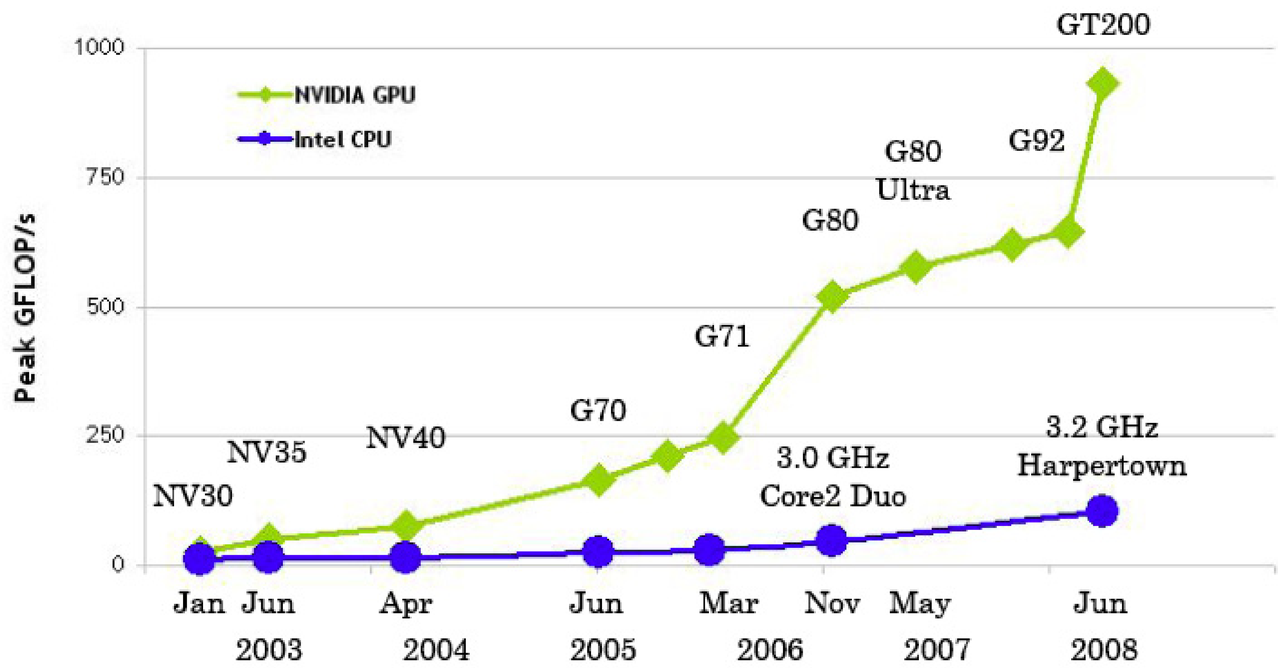


(sequence 0123456789 at 17,387,594,880th digit of pi)

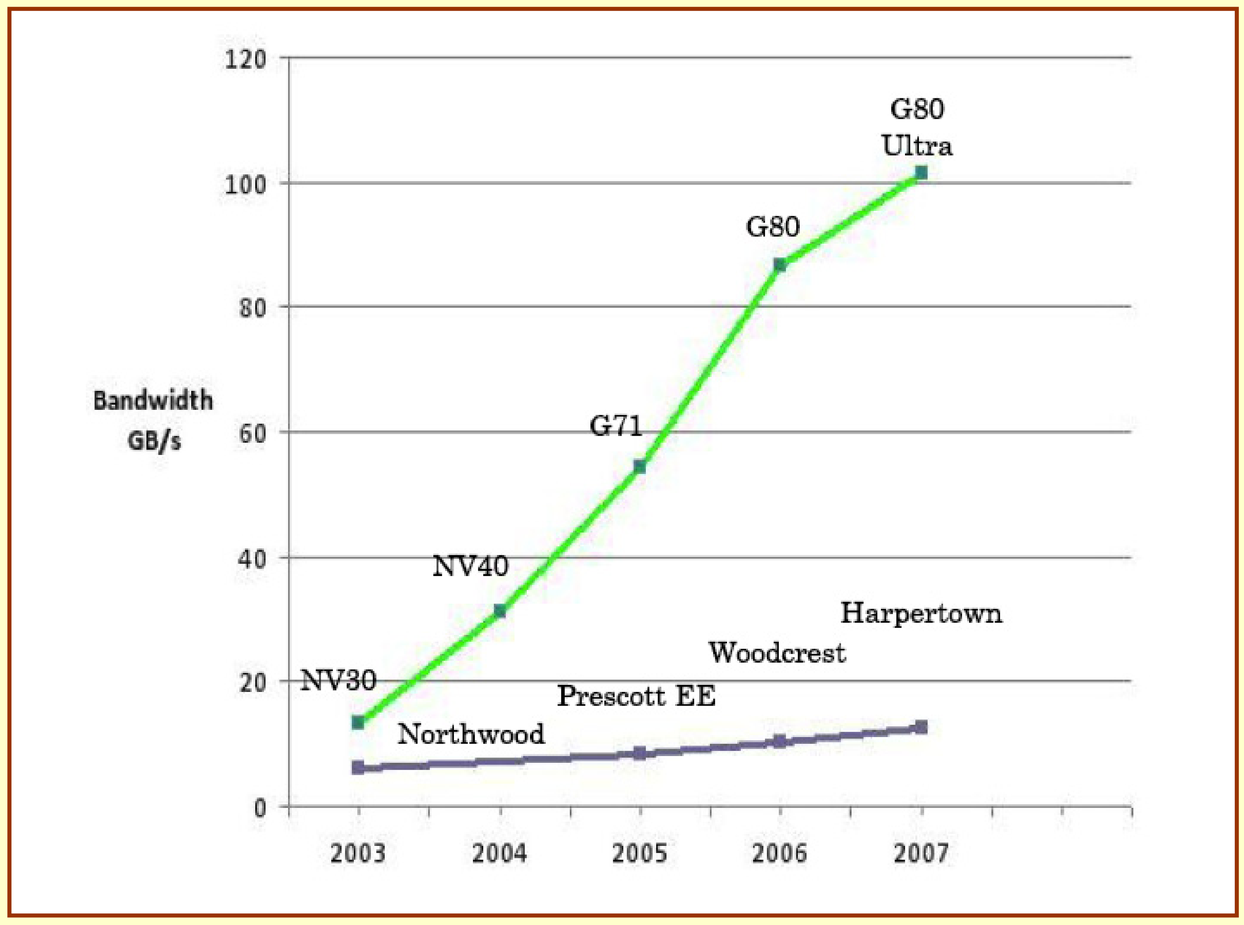
PC Games cards (GPU)

- Cutting edge in last 10 years
- Possible to use for computation but hard
- Became easy in 2006 with NVIDIA CUDA tookit.

GPUs



GPUs – Also Memory Bandwidth



Available Resources

- Own PC (perhaps multi-core and with GPU)
- BSS server (8 cores + Tesla GPU)
- Camgrid (ask Owen)
- Cambridge HPC ask me
- National Centre
- etc...

Programming Recommendations

- MPI (message passing interface) for conventional cores
  - portable
  - well supported and documented
  - introduces idea of *collective operations*
- CUDA for NVIDIA GPUs
  - in many ways similar but with global memory space and limited communication
  - scales to huge numbers of threads
  - cluster of multiprocessors

MPI

- Function Library
- Same program code runs on all cores
- 100's of functions but you only need a tiny number
- Rarely need the send/receive calls used in most introductions!
- Learn the collective operations.

Hello World

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  pri
  ret
}
```

MPI Header

node specifies which core I am out of a total of nodes. Typically use to coordinate calculation.

eg if(node==0) printf(“stuff\n”);

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
  int node, nodes;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &node);

  printf("Hello mpi world, I am node %d out of %d in all\n", node,nodes);

  MPI_Finalize();
  return 0;
}
```

standard initialisation

standard close

Run using MPICH2

```
Command Prompt

Z:\code08\democodes\Debug>democodes.exe
Hello mpi world, I am node 0 out of 1 in all

Z:\code08\democodes\Debug>"c:\Program Files\MPICH2"\bin\mpiexec -n 4 democodes.exe
Hello mpi world, I am node 1 out of 4 in all
Hello mpi world, I am node 0 out of 4 in all
Hello mpi world, I am node 3 out of 4 in all
Hello mpi world, I am node 2 out of 4 in all

Z:\code08\democodes\Debug>"c:\Program Files\MPICH2"\bin\mpiexec -n 16 democodes.exe
Hello mpi world, I am node 4 out of 16 in all
Hello mpi world, I am node 6 out of 16 in all
Hello mpi world, I am node 3 out of 16 in all
Hello mpi world, I am node 2 out of 16 in all
Hello mpi world, I am node 14 out of 16 in all
Hello mpi world, I am node 10 out of 16 in all
Hello mpi world, I am node 12 out of 16 in all
Hello mpi world, I am node 0 out of 16 in all
Hello mpi world, I am node 8 out of 16 in all
Hello mpi world, I am node 5 out of 16 in all
Hello mpi world, I am node 13 out of 16 in all
Hello mpi world, I am node 9 out of 16 in all
Hello mpi world, I am node 7 out of 16 in all
Hello mpi world, I am node 11 out of 16 in all
Hello mpi world, I am node 15 out of 16 in all
Hello mpi world, I am node 1 out of 16 in all

Z:\code08\democodes\Debug>
```

Matrix Multiply Example

$$\mathbf{C=AB}$$

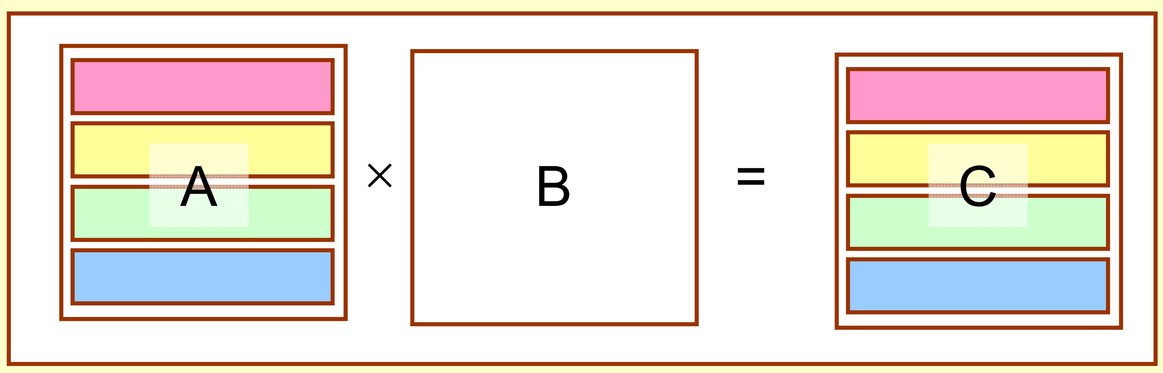
$$C_{ij}=\sum_{k=1}^{N} A_{ik}B_{kj}$$

for N x N matrices need N^3 multiply-adds (mad)

Share rows of A across nodes.

Full copy of B required on all nodes

But only N/nodes rows of A and C required on one node



MatMul Declarations

```
// simple matrix multipy on host
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>

typedef struct {
   int cols;
   int rows;
   int stride;
   float* m;
} Matrix;

// prototypes
void hostMatMul(const Matrix A, const Matrix B, Matrix C);
void randomInit(float* data, int size);
clock_t watch (int it);
```

MatMul Main

```
int main(int argc,char *argv[])
{
      if(argc <2) {
            printf("usage simpleMatMul <size>\n");
            return 0;
      }
      watch(-1);  // start timers
      watch(1);
      Matrix A; Matrix B;   Matrix C;

      int msize = 16*atoi(argv[1]);

      A.cols = A.rows = A.stride = msize;
      B.cols = B.rows = B.stride = msize;
      C.cols = C.rows = C.stride = msize;

      A.m = new float[msize*msize]; randomInit(A.m,msize*msize);
      B.m = new float[msize*msize]; randomInit(B.m,msize*msize);
      C.m = new float[msize*msize];

      hostMatMul(A,B,C);

      printf("Timings for C=A*B for %d x %d matricies\n",msize,msize);
      watch(-2);

      return 0;
}
```


Mat Mul Code - Host

```
void hostMatMul( Matrix A,Matrix B,Matrix C )
{
     watch(2);
     for(int row=0;row<A.cols;row++) for(int col=0;col<B.rows;col++){
          float sum=0.0f;
          for(int k=0;k<A.cols;k++) sum += A.m[row*A.cols+k]*B.m[k*B.cols+col];
          C.m[row*A.cols+col] = sum;
     }
     return;
}
```

Main with MPI

```
int main(int argc,char *argv[])
{
    int node=0;  int nodes=1; watch(-1);
    if (MPI_Init(&argc, &argv) == MPI_SUCCESS) {
        MPI_Comm_size (MPI_COMM_WORLD, &nodes);
        MPI_Comm_rank (MPI_COMM_WORLD, &node );
        if(node==0 && nodes > 1)printf("MPI for %d nodes\n",nodes);
    }

    Matrix A;  Matrix B;  Matrix C;
    int msize = 16*atoi(argv[1]);
    A.cols = A.rows = A.stride = msize;  B.cols = B.rows = B.stride = msize;
    C.cols = C.rows = C.stride = msize;

    if(node==0){
        A.m = new float[msize*msize];   randomInit(A.m,msize*msize);
        B.m = new float[msize*msize];   randomInit(B.m,msize*msize);
        C.m = new float[msize*msize];
        hostMatMul(A,B,C);
    }

    mpiMatMul(A,B,C,node,nodes);
    MPI_Barrier(MPI_COMM_WORLD);  // wait for all to finish
    if(node==0){
        printf("Timings for C=A*B for %d x %d matricies\n",msize,msize);
        watch(-2);
    }

    MPI_Finalize();
    return 0;
}
```

Initialize MPI

Create A, B & C ONLY on node 0

Parallel multiply

Wait for all nodes

Print on node 0

MPI version

```
void mpiMatMul( Matrix A, Matrix B, Matrix C, int node, int nodes)
{
    watch(3);

    A.rows /= nodes;  // care here node 0 IS full size
    C.rows /= nodes;
    int msize = A.rows*A.cols; // submatrix size

    // create local buffers for only subset of rows of A and C to be stored
    if(node>0){
        A.m = new float[A.rows*A.cols];  B.m = new float[B.rows*B.cols]; C.m = new float[C.rows*C.cols];
    }

    MPI_Scatter(A.m,msize,MPI_FLOAT,A.m, msize,MPI_FLOAT,0,MPI_COMM_WORLD);
    MPI_Bcast( B.m,B.rows*B.cols,MPI_FLOAT,0,MPI_COMM_WORLD);

    for(int row=0;row<A.rows;row++) for(int col=0;col<B.rows;col++){
        float sum = 0.0f;
        for(int k=0;k<A.cols;k++) sum += A.m[row*A.cols+k]*B.m[k*B.cols+col];
        C.m[row*A.cols+col] = sum;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(A.m,msize,MPI_FLOAT,A.m, msize,MPI_FLOAT,0,MPI_COMM_WORLD);

    if(node>0){
        delete[] A.m; delete[] B.m; delete[] C.m;
    }
    return;
}
```

Allocate A, B & C

Distribute A between all

Copy B to All

Wait for All Done

Tidy UP

Copy all blocks to Node 0

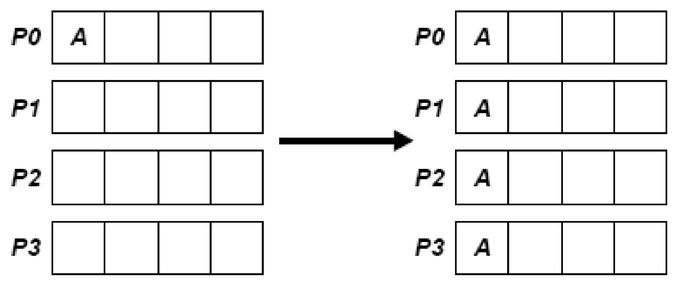
Test on old 2-core PC

Command Prompt

```
Z:\code08\democodes\Release>matmul.exe 32
Timings for C=A*B for 512 x 512 matricies
timer Other   Total     0 ms, calls= 1, per call 0.000000
timer Setup   Total    16 ms, calls= 1, per call 16.000000
timer Host    Total  1297 ms, calls= 1, per call 1297.000000
Total time 1.313 secs

Z:\code08\democodes\Release>matmul.exe 60
Timings for C=A*B for 960 x 960 matricies
timer Other   Total     0 ms, calls= 1, per call 0.000000
timer Setup   Total    93 ms, calls= 1, per call 93.000000
timer Host    Total  8969 ms, calls= 1, per call 8969.000000
Total time 9.062 secs

Z:\code08\democodes\Release>"c:\Program Files\MPICH2"\bin\mpiexec -n 2 matmul.exe 32
MPI for 2 nodes
Timings for C=A*B for 512 x 512 matricies
timer Other   Total     0 ms, calls= 1, per call 0.000000
timer Setup   Total    16 ms, calls= 1, per call 16.000000
timer Host    Total  1281 ms, calls= 1, per call 1281.000000
timer MPI     Total   640 ms, calls= 1, per call 640.000000
Total time 1.937 secs

Z:\code08\democodes\Release>"c:\Program Files\MPICH2"\bin\mpiexec -n 2 matmul.exe 60
MPI for 2 nodes
Timings for C=A*B for 960 x 960 matricies
timer Other   Total     0 ms, calls= 1, per call 0.000000
timer Setup   Total    94 ms, calls= 1, per call 94.000000
timer Host    Total  9047 ms, calls= 1, per call 9047.000000
timer MPI     Total  4750 ms, calls= 1, per call 4750.000000
Total time 13.891 secs

Z:\code08\democodes\Release>
```


Broadcast (copy to all)

- Broadcasting:

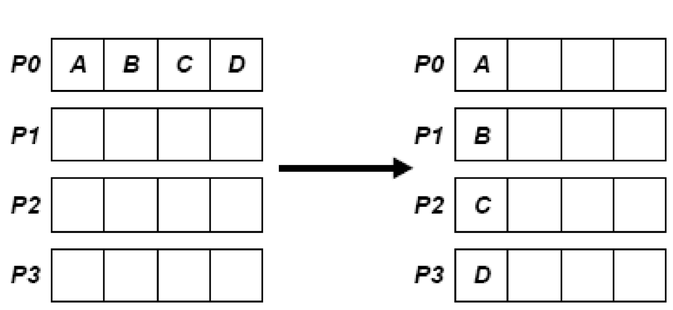



```
int MPI_Bcast(void* buffer, int count,
  MPI_Datatype datatype, int root,
  MPI_Comm comm );
```

NB MPI can be smart, using switch optimally

Scatter – share data between nodes

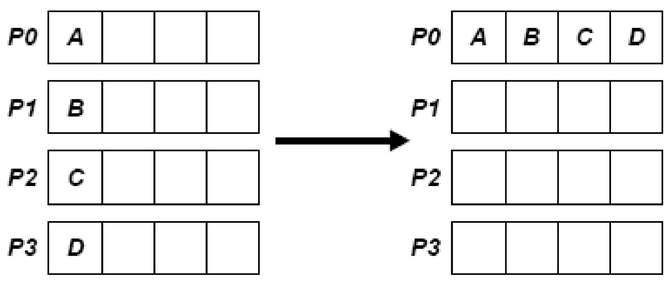
- Scatter operation:




```
int MPI_Scatter(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf,
  int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm);
```


Gather – collect data to one node

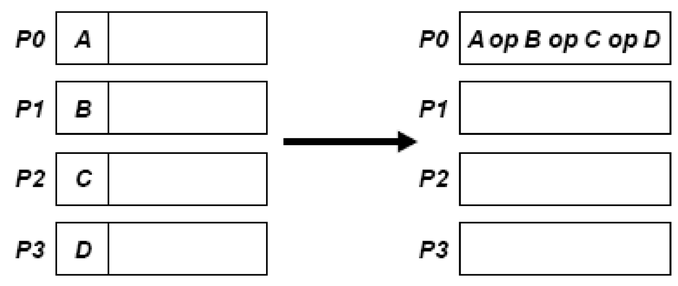
- Gather operation:




```
int MPI_Gather(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf,
  int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm);
```

Reduce, e.g. sum across nodes

- **Reduce operation:**




```
int MPI_Reduce(void* sendbuf, void* recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm);
```

- Useful ops include `MPI_SUM`, `MPI_PROD`, `MPI_MIN` and `MPI_MAX`.
- Can also define your own operations.

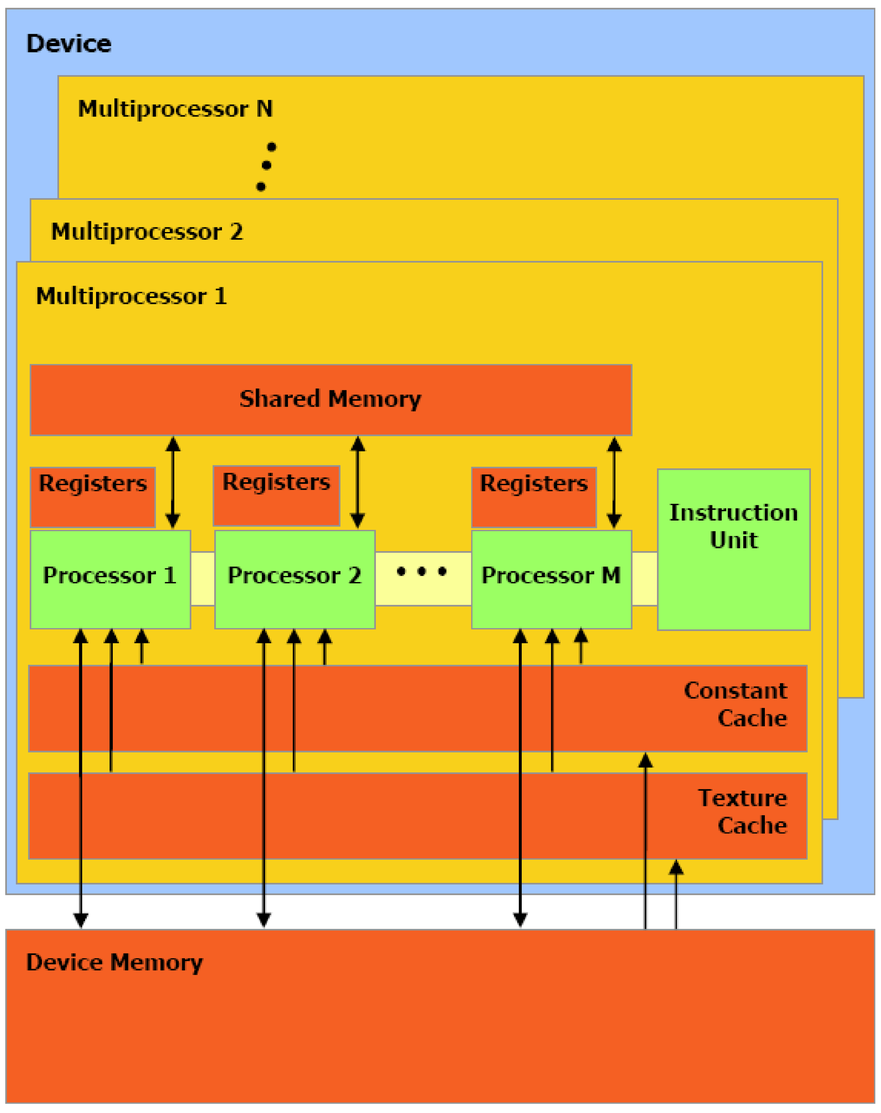
Others

Explicit synchronisation

- **Barrier synchronization:**
  `int MPI_Barrier(MPI_Comm comm);`
- **Timing your program:**
  `double MPI_Wtime();`

CUDA- Recent NVIDIA GPUs

- Cards have up to 30 multiprocessors
- Each multiprocessor has 8 cores
- Each multiprocessor runs a thread block of up to 512 threads “simultaneously”
- Thread blocks can share limited fast memory
- No communication between thread blocks
- CUDA runs sufficient tread blocks to complete task



CUDA Device

Shared memory is local to multiprocessor and is not visible to host. Shared memory does not persist

The other memories are global and are visible to host and are persistent across multiple kernel executions.

SIMD model.

CUDA Device Query (st1)

Untitled - Notepad

File Edit Format View Help

```
CUDA Device Query (Driver API) statically linked version
There are 2 devices supporting CUDA

Device 0: "Tesla C1060"
  CUDA Driver Version:                           2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads
can use this device simultaneously)


Test PASSED

Press ENTER to exit...
```

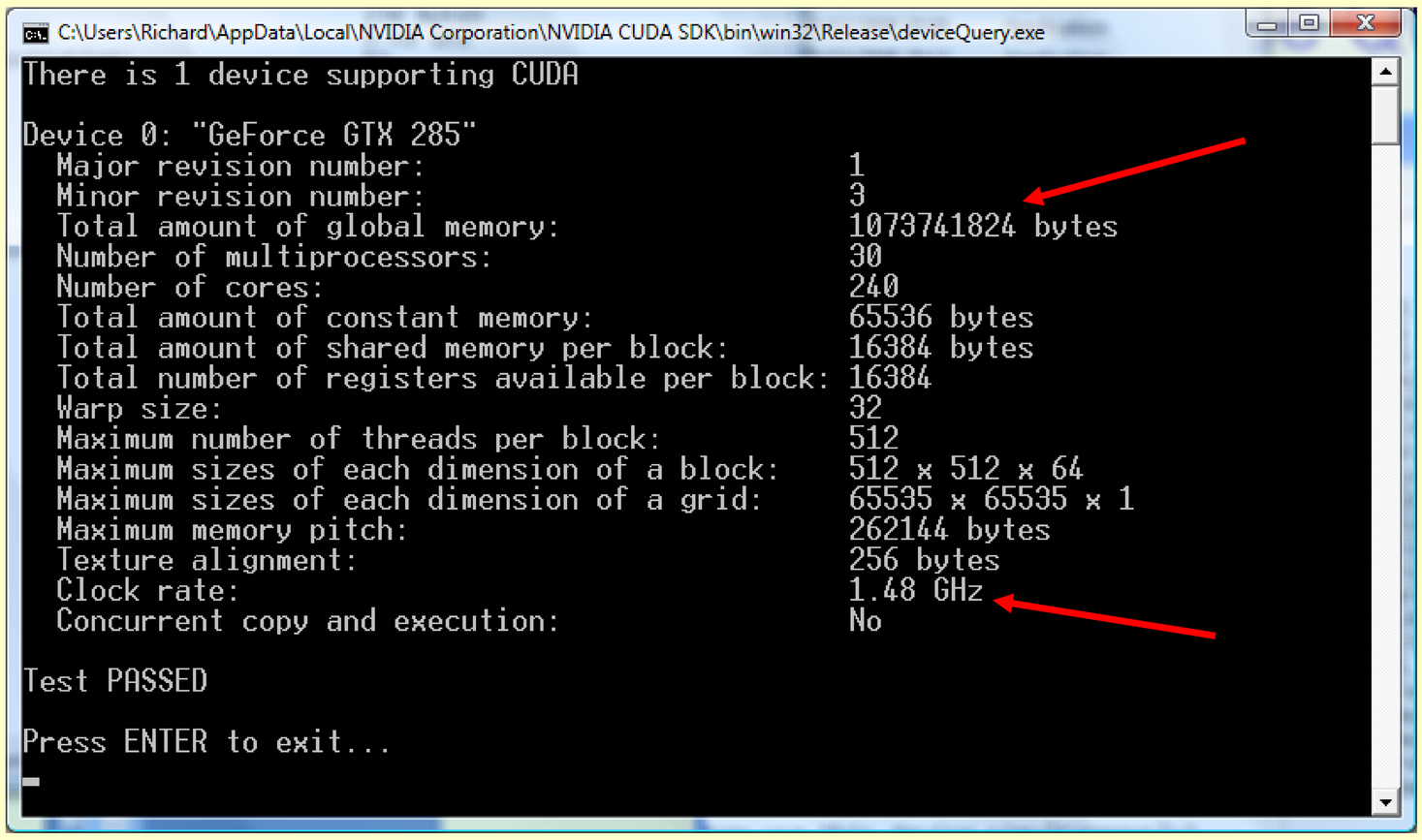

CUDA Device Query (my BSS PC)

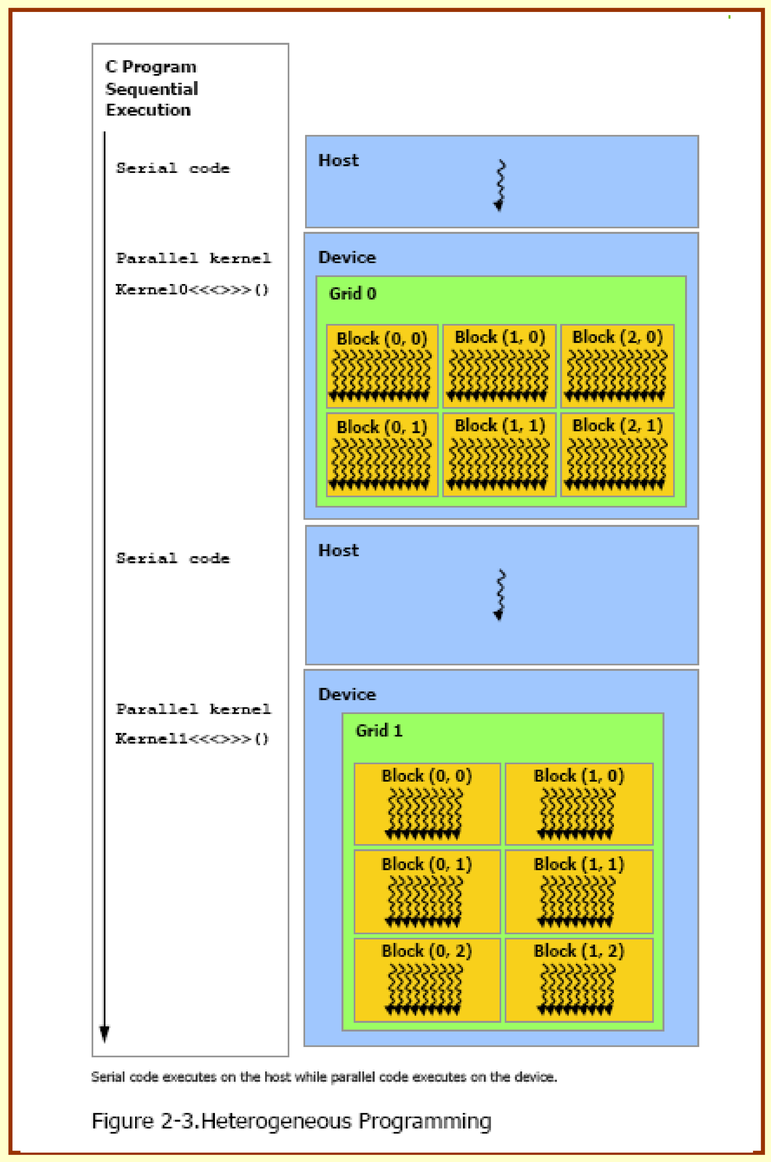
Untitled - Notepad

File Edit Format View Help

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "GeForce 9600 GT"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 1073414144 bytes
  Number of multiprocessors:                     8
  Number of cores:                               64
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.65 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       No
  Compute mode:                                  Default (multiple host threads
can use this device simultaneously)

Test PASSED

Press ENTER to exit...
```

CUDA Device Query (Games PC)





Serial code executes on the host while parallel code executes on the device.

Figure 2-3.Heterogeneous Programming

CUDA Program

CUDA program runs on single host core. Typically:

- sends data to device memories.
- launches one or more kernels on device.
- collects final results back from device global memory.
- A kernel is a piece of C like code which is run by many threads.

CUDA Matrix Multiply first version

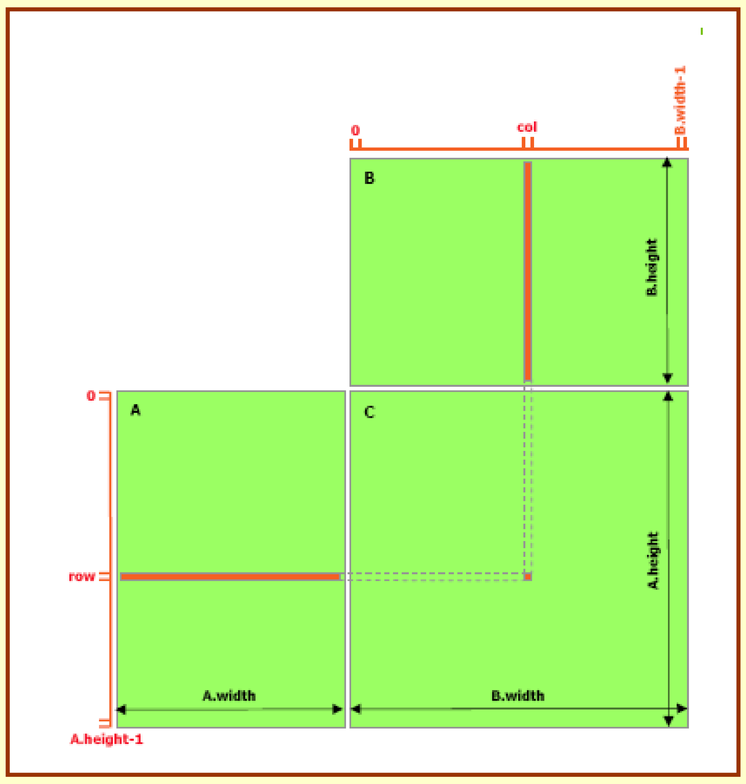
Use separate threads for each element C_{ij} in matrix product

C= A*B

Thus N^2 threads.

Each thread block will correspond to a 16x16 sub matrix of C.

(NB 16x16 is often a very good choice)



Kernel Code

local variable in register thus fast

Find which element of C this thread needs to calculate. The variables on RHS are maintained by CUDA

Same function arguments! Set by Host but data pointer is to device memory

A, B and C are allocated by Host in device global memory – thus slow.

```
// Matrix multiplication kernel called by devMatrixMul()
__global__ void MatMulKernel( Matrix A, Matrix B, Matrix C )
{
    // Each thread computes one element of C accumulating results into sum
    float sum = 0.0f;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col  = blockIdx.x * blockDim.x + threadIdx.x;

    for (int k=0;k<A.cols;++k) sum += A.m[row*A.cols+k]*B.m[k*B.cols+col];
    C.m[row*C.cols+col] = sum;
}
```

```
void hostMatMul( Matrix A,Matrix B,Matrix C )
{
    watch(2);
    for(int row=0;row<A.cols;row++) for(int col=0;col<B.rows;col++){
        float sum=0.0f;
        for(int k=0;k<A.cols;k++) sum += A.m[row*A.cols+k]*B.m[k*B.cols+col];
        C.m[row*A.cols+col] = sum;
    }
    return;
}
```

```
void devMatMul(Matrix A, Matrix B, Matrix C)
{
    // Copy A and B to devA and devB in device memory
    Matrix devA;
    devA.cols = A.cols; devA.rows = A.rows; devA.stride = A.stride;
    size_t size = A.cols * A.rows * sizeof(float);
    cudaMalloc((void**)&devA.m, size);
    cudaMemcpy( devA.m, A.m, size ,cudaMemcpyHostToDevice);

    Matrix devB;
    devB.cols = B.cols; devB.rows = B.rows;  devB.stride = B.stride;
    size = B.cols * B.rows * sizeof(float);
    cudaMalloc((void**)&devB.m, size);
    cudaMemcpy( devB.m, B.m, size ,cudaMemcpyHostToDevice);

    // Allocate devC in device memory no need to copy
    Matrix devC;
    devC.cols = C.cols; devC.rows = C.rows;  devC.stride = C.stride;
    size = C.cols * C.rows * sizeof(float);
    cudaMalloc((void**)&devC.m, size);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(B.cols/dimBlock.x, A.rows/dimBlock.y);

    // Invoke kernel
    MatMulKernel<<<dimGrid, dimBlock>>>(devA, devB, devC);
    cudaThreadSynchronize();
    // copy result back to host
    cudaMemcpy( C.m, devC.m, size, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    // Free device memory
    cudaFree(devA.m); cudaFree(devB.m);  cudaFree(devC.m);
}
```

CUDA host code

Create copies of A, B & C in device memory

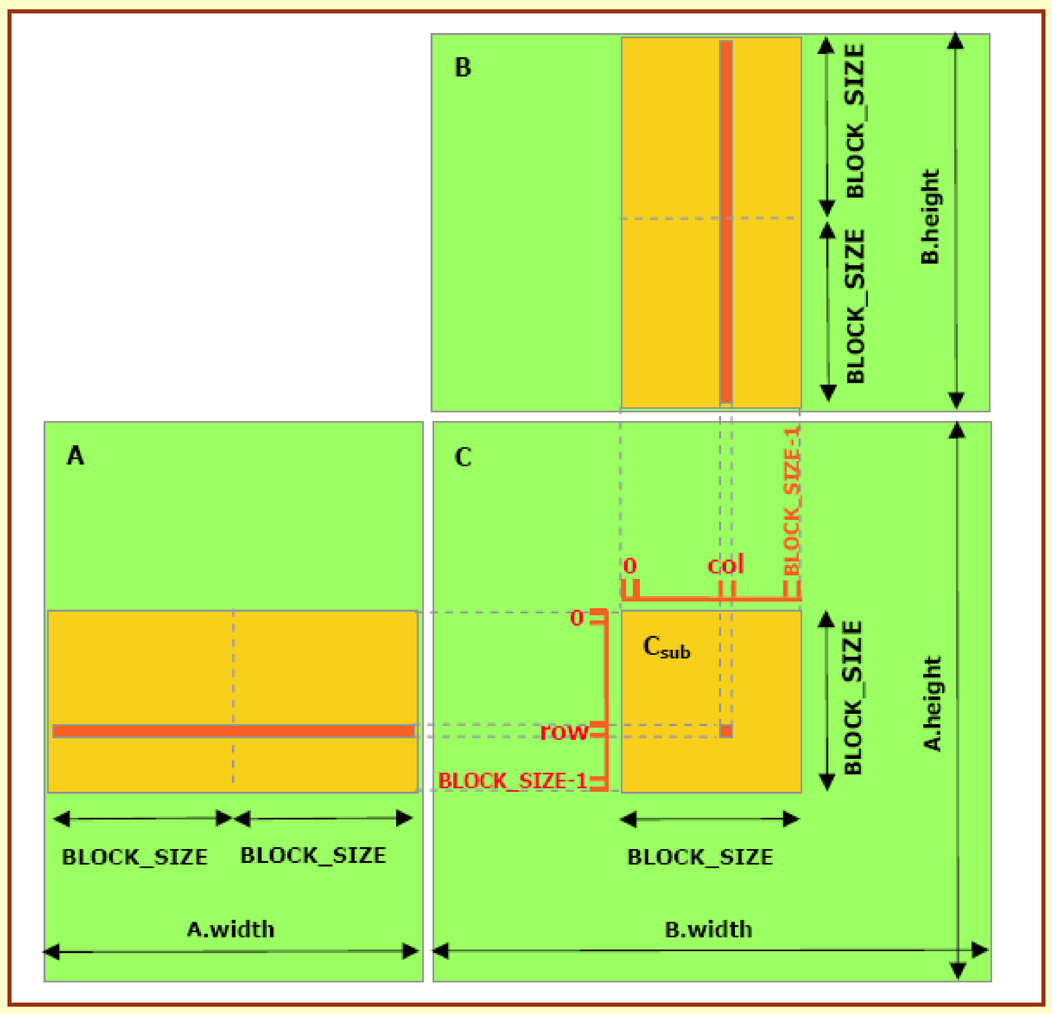
define thread blocks

CUDA kernel call, note arguments

Copy answer back to host

Faster kernel using shared memory

Each thread block uses shared memory to cache a 16x16 sub-blocks of the matrices. This improves performance, especially when addressing B with a large stride.



Faster kernel using shared memory

```
__global__ void MatMulSmemKernel( Matrix A, Matrix B, Matrix C )
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread in block computes one element of Csub, accumulating result into sum
    int row = threadIdx.y;
    int col = threadIdx.x;

    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // Loop over all the sub-matrices of A and B that are required to compute Csub
    float sum = 0.0f;
    for (int m = 0; m < (A.cols/16); ++m) {
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        As[row][col] = GetElement( Asub, row, col);
        Bs[row][col] = GetElement( Bsub, row, col);
        __syncthreads();

        // Multiply Asub and Bsub
        for (int k=0;k<16;++k) sum += As[row][k]*Bs[k][col];
        __syncthreads();
    }
    // Write Csub to device memory, each thread writes one element
    SetElement(Csub, row, col, sum);
}
```

declare shared memory

Loop over parts

Cache A &B sub matrices

evaluate this sub-matrix contribution

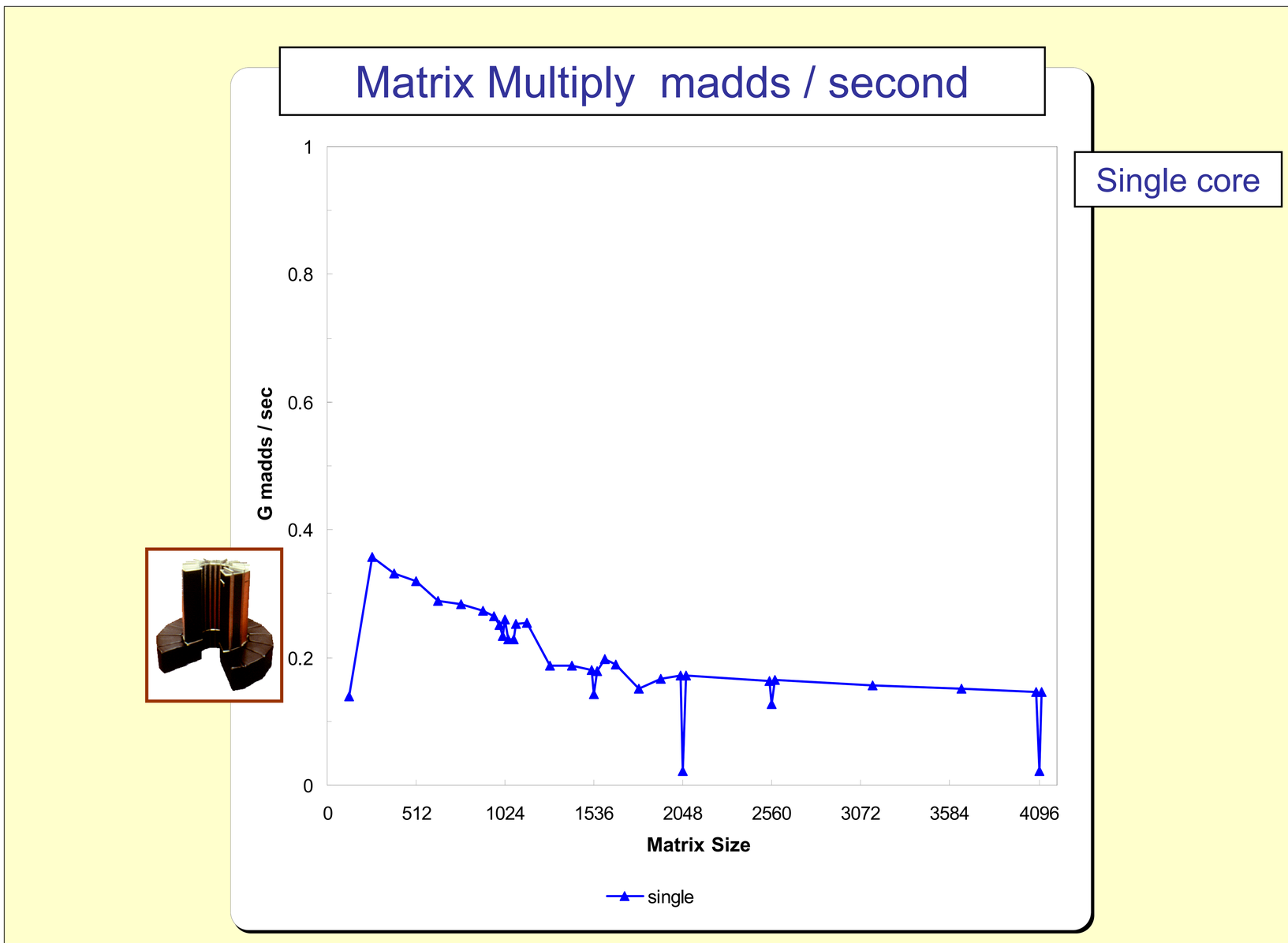
Support functions

```
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
        return A.m[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement( Matrix A, int row, int col,float value)
{
        A.m[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
// BLOCK_SIZE is #defined as 16
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
        Matrix Asub;
        Asub.cols = BLOCK_SIZE;
        Asub.rows = BLOCK_SIZE;
        Asub.stride = A.stride;
        Asub.m = &A.m[A.stride * BLOCK_SIZE * row+ BLOCK_SIZE * col];
        return Asub;
}
```

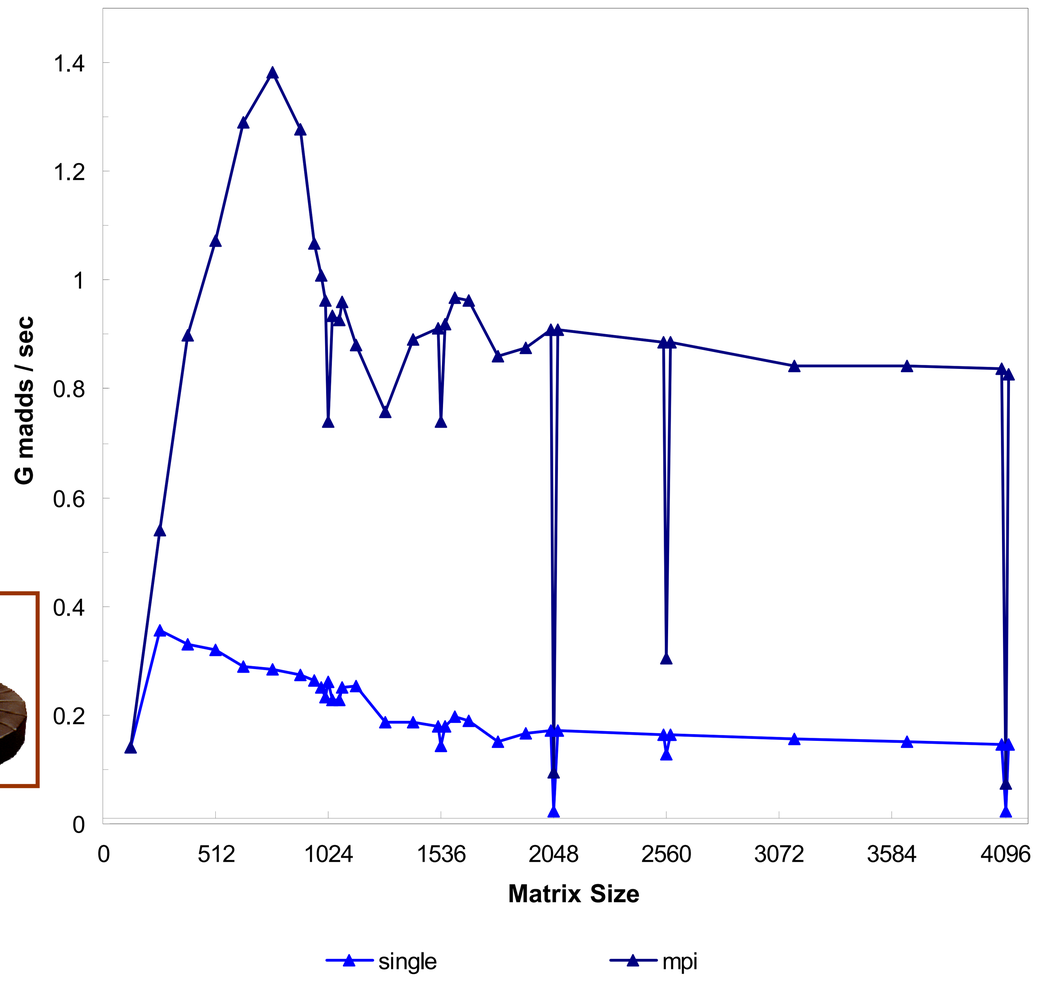
Results

Command Prompt

```
Z:\cudaapp\Release>"c:\Program Files\MPICH2\bin\mpiexec.exe" -n 8 simpleMatMul.exe 63
MPI for 8 nodes
Timings for C=A*B for 1008 x 1008 matricies
timer Other    Total    63 ms, calls= 1, per call 63.000000
timer Host     Total  4407 ms, calls= 1, per call 4407.000000
timer MPI      Total  1078 ms, calls= 1, per call 1078.000000
timer CUDAini Total    62 ms, calls= 2, per call 31.000000
timer Kernel1 Total   156 ms, calls= 2, per call 78.000000
timer Kernel2 Total    94 ms, calls= 10, per call 9.400000
timer memcpy1 Total     0 ms, calls= 2, per call 0.000000
timer memcpy2 Total    31 ms, calls= 10, per call 3.100000
Total time 5.891 secs

Z:\cudaapp\Release>"c:\Program Files\MPICH2\bin\mpiexec.exe" -n 8 simpleMatMul.exe 100
MPI for 8 nodes
Timings for C=A*B for 1600 x 1600 matricies
timer Other    Total   141 ms, calls= 1, per call 141.000000
timer Host     Total 20859 ms, calls= 1, per call 20859.000000
timer MPI      Total  4344 ms, calls= 1, per call 4344.000000
timer CUDAini Total    62 ms, calls= 2, per call 31.000000
timer Kernel1 Total   625 ms, calls= 2, per call 312.500000
timer Kernel2 Total   422 ms, calls= 10, per call 42.200000
timer memcpy1 Total     0 ms, calls= 2, per call 0.000000
timer memcpy2 Total    47 ms, calls= 10, per call 4.700000
Total time 26.500 secs

Z:\cudaapp\Release>_
```

Matrix Multiply madds / second

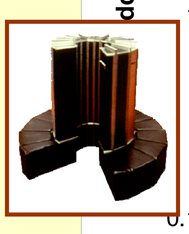


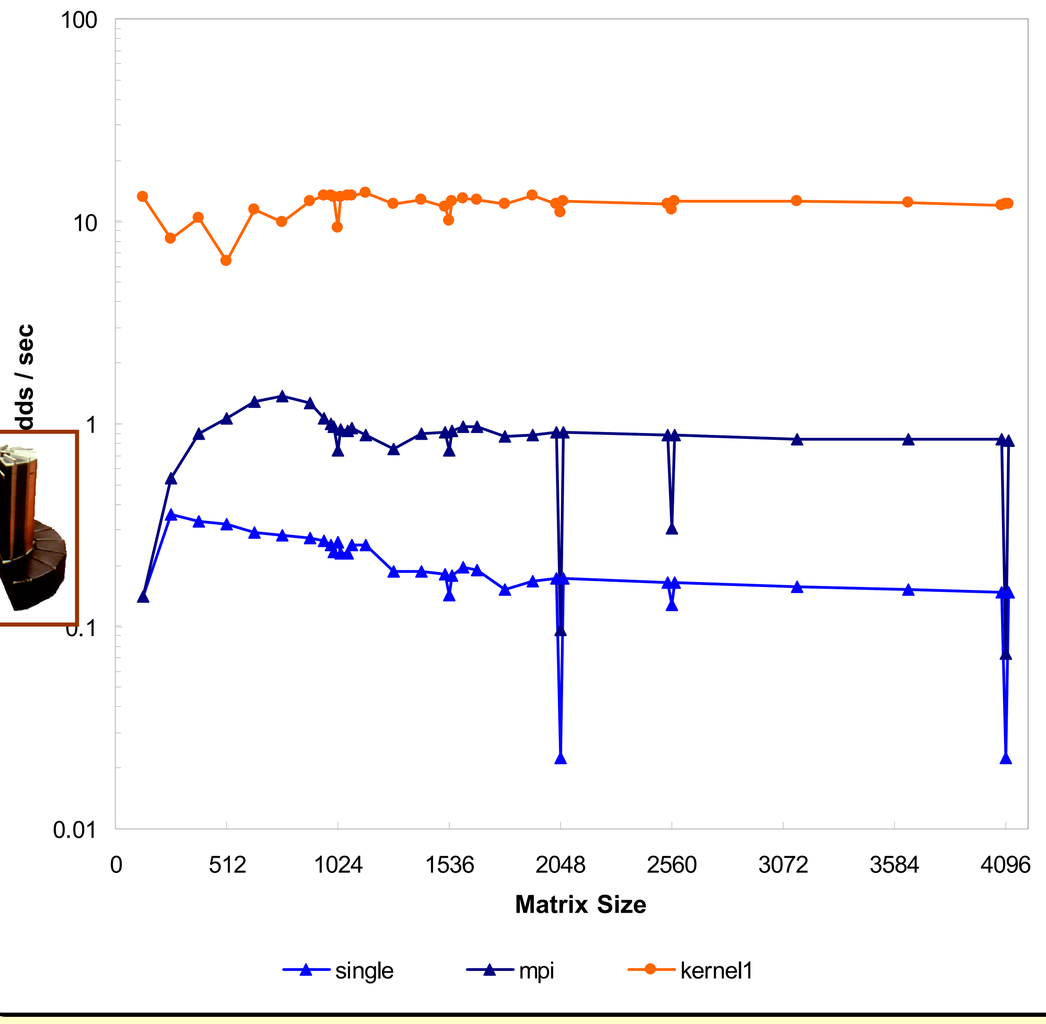
Single core

Speedup

8 cores
MPI: 5.2

Matrix Multiply madds / second





Single core

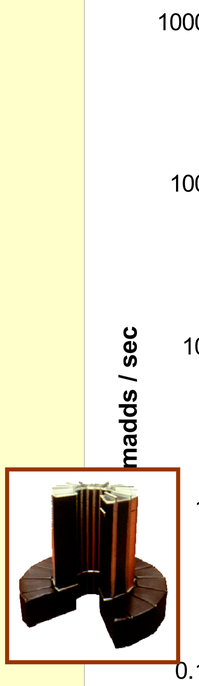
Speedups

8 cores
MPI: 5.2

Kernel 1

x 71.6

Matrix Multiply madds / second



Single core

Speedups

8 cores
MPI:x 5.2

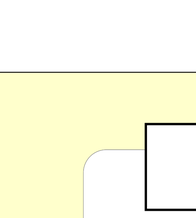
Kernel 1

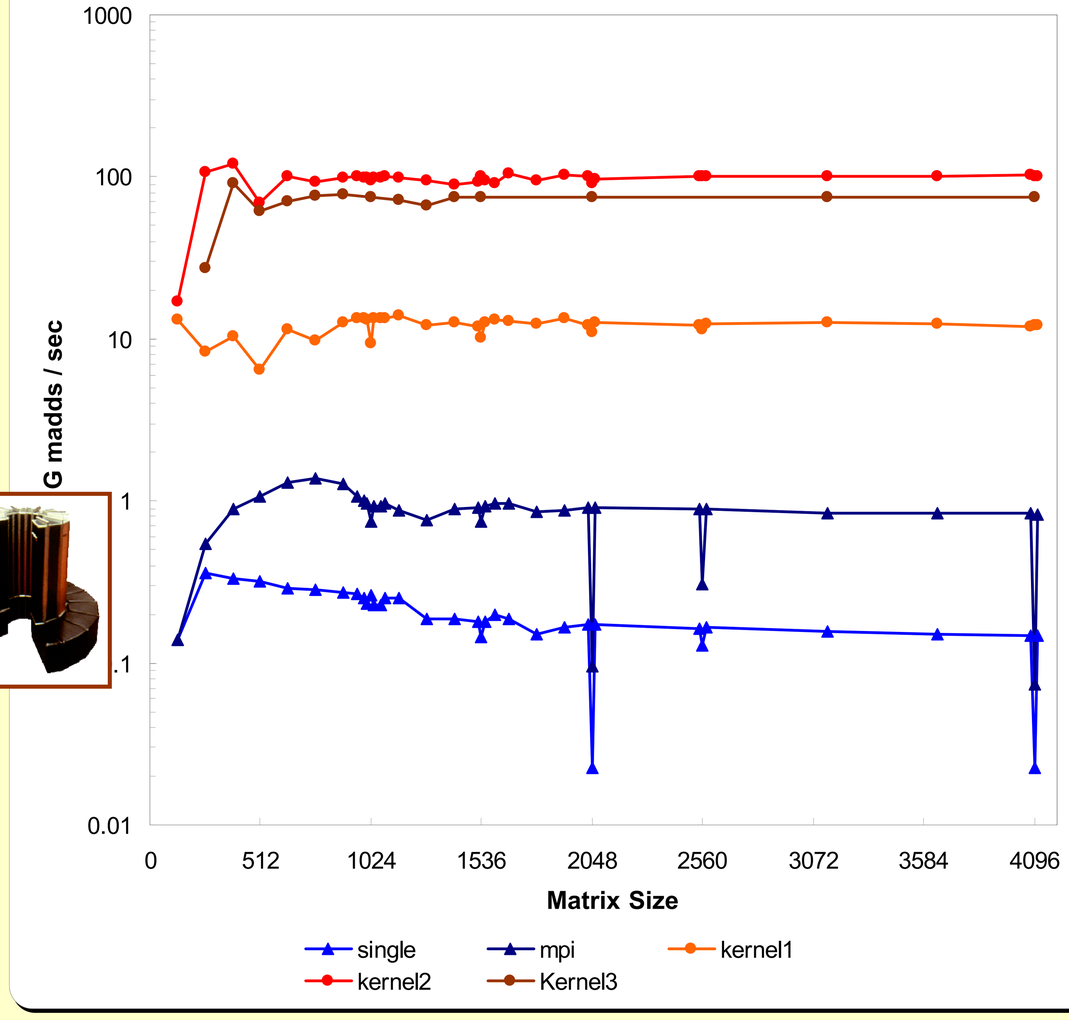
x 71.6

Kernel 2

x 577

Matrix Multiply madds / second





Single core

Speedups

8 cores
MPI:x 5.2

Kernel 1

x 71.6

Kernel 2

x 577

Kernel 3

x 475

References:

NVIDIA CUDA: http://www.nvidia.com/cuda

MPI(MPICH2):
http://www.mcs.anl.gov/research/projects/mpich2/

Cambridge:

- http://www.hpc.cam.ac.uk/
- http://www.escience.cam.ac.uk/projects/camgrid/
- http://www.many-core.group.cam.ac.uk/

GPGPU: http://gpgpu.org/

Magnetic Resonance

NMR – Spectroscopy

MRI- Imaging

NMR History

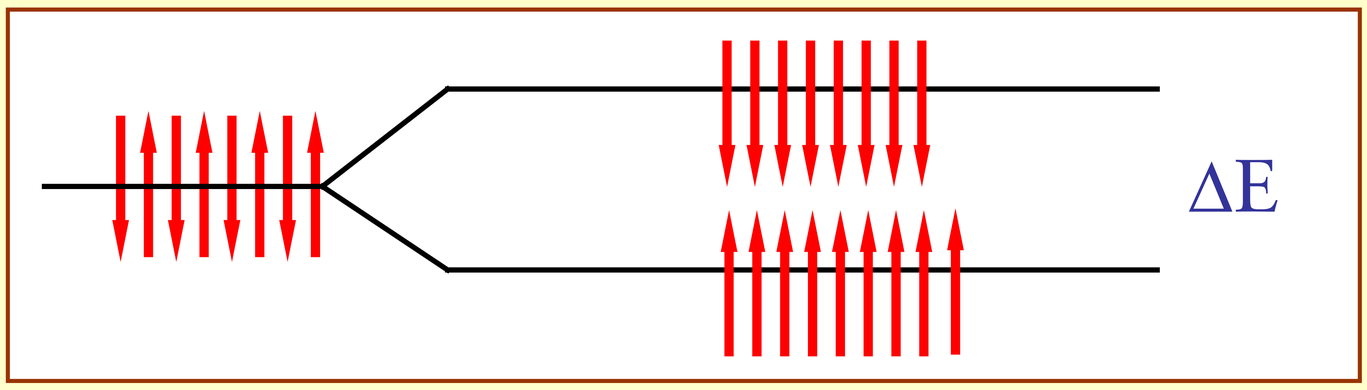
- 1921: Compton: electron spin
- 1924: Pauli: Proposes nuclear spin
- 1946: Stanford/Harvard group detect first NMR signal
- mid -50 to mid 70's NMR become powerful tool for structural analysis
- mid-70 first superconducting magnets

MRI History

- 1973: Lauterbur: First NMR image of sample tubes in a chemical spectrometer
- 1981: First commercial scanners <0.2T
- 1985: 1.5T scanner
- 1986: Rapid developments in SNR, resolution etc
- 1998: Whole body 8T at OSU
- 2003: Nobel Prize for Peter Mansfield & Paul Lauterbur

Nuclear Zeeman Effect

Application of strong magnetic field B_0 lifts degeneracy of nuclear spin levels



For spin 1/2: $\Delta E = \bar{\gamma}\, h\, B_0$

γ Gyromagnetic ratio (constant of nucleus)

For hydrogen γ = 42.5 MHz/T

Population Difference

Given by Boltzman Statistics:

$$\frac{N_+}{N_-} = e^{-\gamma\hbar B_0/kT}$$

population difference is small <1 in 10^6

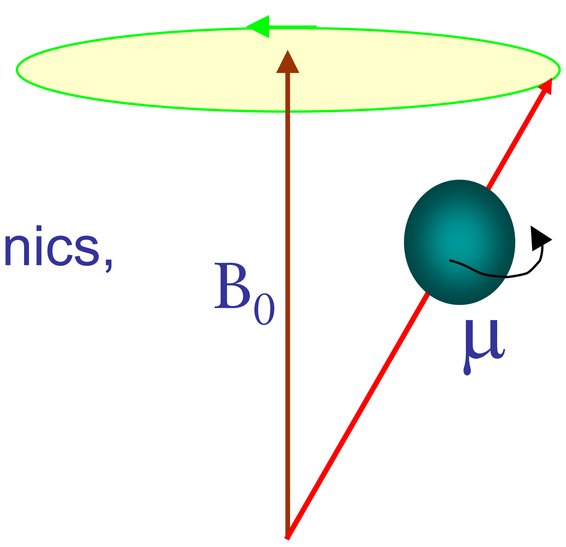
NMR/MRI is very insensitive

Semi-Classical Model

Gyroscopic motion of magnetic moment about B_0

Use classical mechanics,
Larmor precession

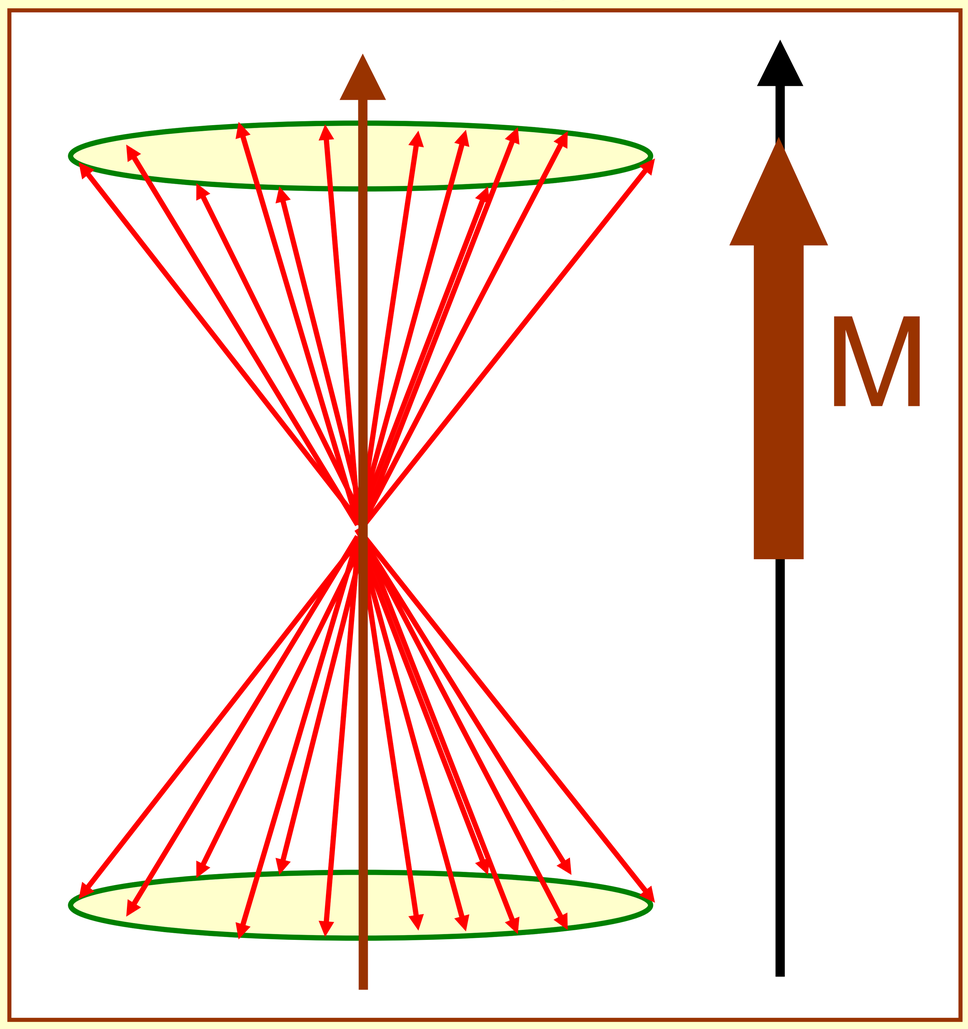
$$\omega_0 = -\gamma B_0$$



Ensemble Average

USE Classical language for behaviour of net magnetization

Same results as quantum treatment.



Spin Precession is like a Gyroscope

Gravity plays the role of the magnetic field for gyroscope .

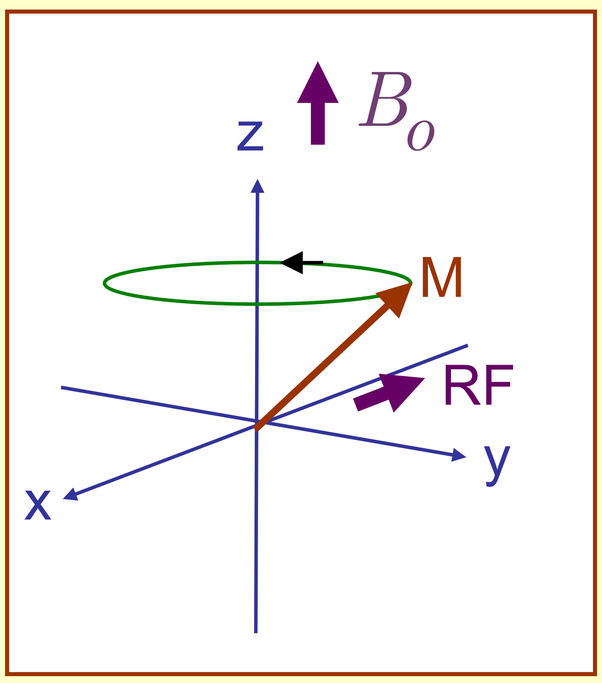
Gyroscope wants to fall but is prevented by conservation of angular momentum.

Similarly *Quantum* bar magnets precess

Effect of RF Pulse

A Radio Frequency pulse at the resonant frequency causes the magnetization vector to rotate away from the Z axis

A 90 degree RF pulse tips the magnetization vector into the x-y plane



Magnetization in the x-y plane radiates RF which is the signal we measure

Magnetic Nuclei

Table 1.1. Gyromagnetic ratio of some nuclei.

Nucleus	^{1}H	^{13}C	^{19}F	^{23}N	^{31}P
$\gamma/2\pi$ (MHz/T)	42.58	10.71	40.08	11.27	17.25

Bloch Equations

$$\frac{dM_{x,y}}{dt} = \gamma(\mathbf{M} \times \mathbf{B})_{x,y} - \frac{M_{x,y}}{T_2}$$
$$\frac{dM_z}{dt} = \gamma(\mathbf{M} \times \mathbf{B})_z + \frac{M_0 - M_z}{T_1}$$

Describe evolution of transverse and longitudinal components of magnetization. T1 and T2 control exponential relaxation and dephasing. T1 and T2 are properties of the material

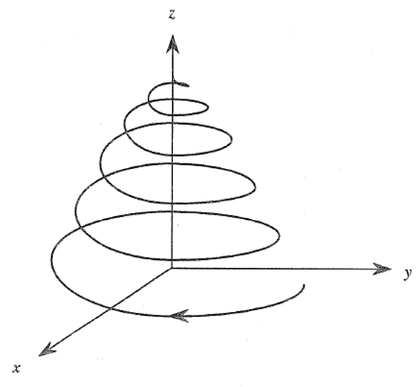
Bloch Equations

$$\frac{dM_{x,y}}{dt} = \gamma(\mathbf{M} \times \mathbf{B})_{x,y} - \frac{M_{x,y}}{T_2}$$

$$\frac{dM_z}{dt} = \gamma(\mathbf{M} \times \mathbf{B})_z + \frac{M_0 - M_z}{T_1}$$

$$\left|M_z(t)\right| = \left|M_z(t=0)\right|(1-e^{-t/T_1})$$

$$\left|M_{x,y}(t)\right| = \left|M_{x,y}(t=0)\right| e^{-t/T_2}$$



Free Induction Decay (FID)

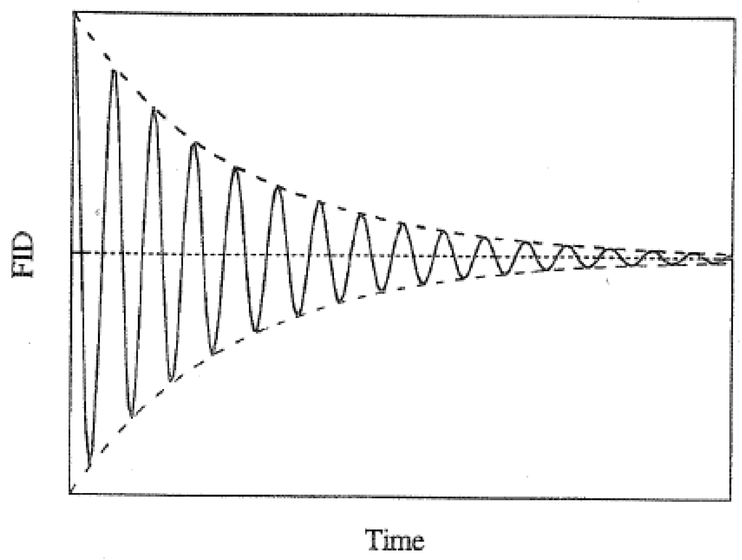
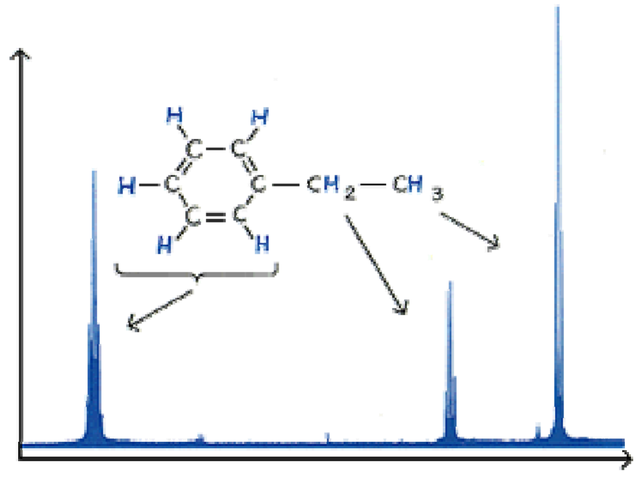


Figure 1.9. The received FID after application of a 90° pulse. Because of relaxation, the received NMR signal exhibits exponential decay.

Free Induction Decay (FID)

Fourier Transform of FID gives spectrum of (say) proton frequencies. These “chemical shifts” plus lots of maths can give important detail for molecular structure.

POM was intended to house a 900 MHz NMR facility



A proton NMR spectrum of a solution containing a simple organic compound, ethyl benzene. Each group of signals corresponds to protons in a different part of the molecule.

Huge importance for chemistry AND molecular biology

Imaging MRI

- Make signals position dependent
- Exploit local T1 and/or T2 variations to get image contrast
- Use additional switched magnetic field gradients to make precession frequencies position and time dependent
- End up with 2D or 3D Fourier Transform of spatial distribution – “k-space image”

Gradients

Signal from sample in volume V at time t is:

$$S(t) = e^{i\omega_0 t} \int_V \rho(\mathbf{r}) e^{i\phi(\mathbf{r},t)} d^3r$$

where ρ is local spin density and ϕ is accumulated local phase offset:

$$\phi(\mathbf{r},t) = \int_0^t (G_x x + G_y y + G_z z) dt = \mathbf{r} \cdot \int_0^t \mathbf{G}\, dt$$

Hence:

$$S(\mathbf{k}) = \int_V \rho(\mathbf{r})\, e^{i\mathbf{k}\cdot\mathbf{r}}\, d^3r \quad \text{where} \quad \mathbf{k} = \int_0^t \mathbf{G}\, dt$$

Gradients are driven to cover the whole of k-space over time and hence map out the FT of ρ.

Spin Echo

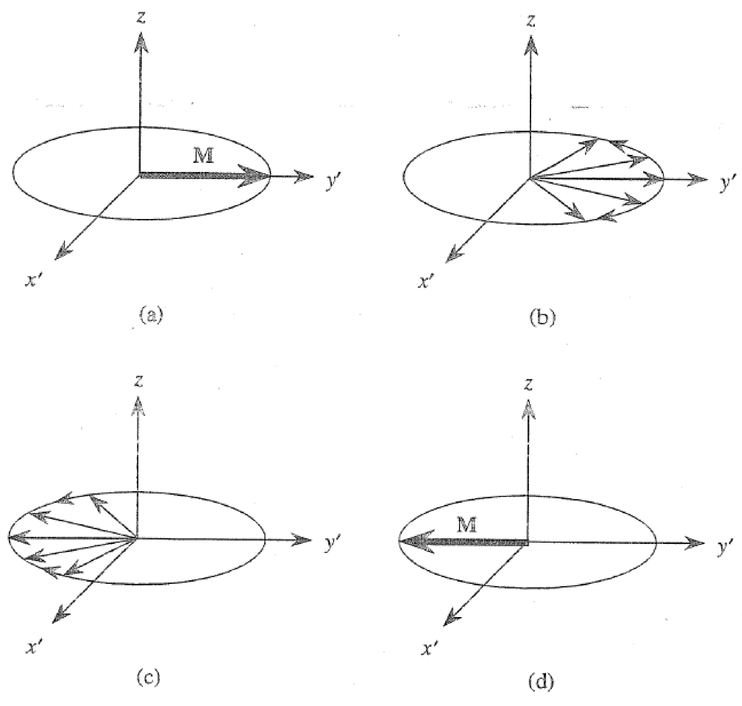


Figure 1.17. (a) After a 90° pulse, the magnetization lies along the y' axis. (b) Because of field inhomogeneity, the nuclei start to lose coherence. (c) Application of a 180° pulse rotates the magnetization. (d) The coherence is re-established.

Pulse sequences Spin Echo

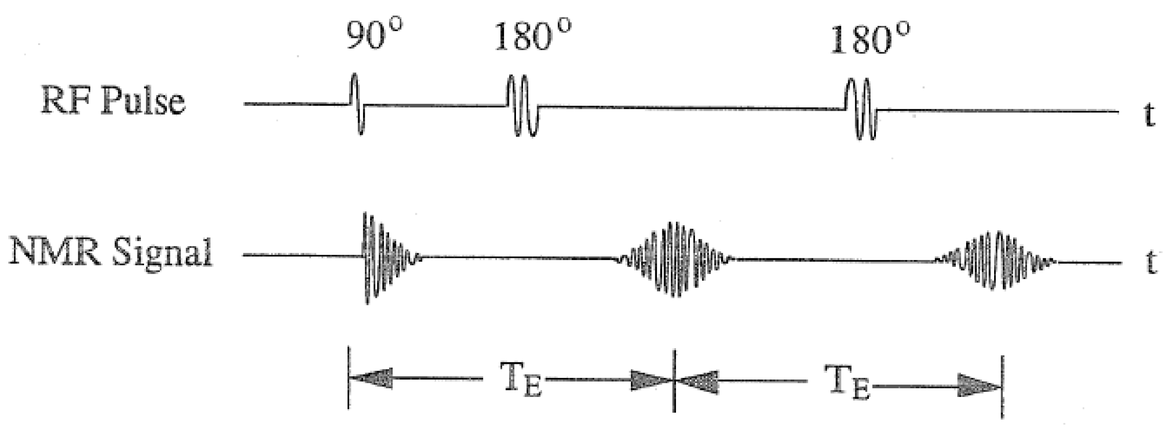


Figure 1.16. Spin echo sequence.

Pulse sequences Gradient Echo

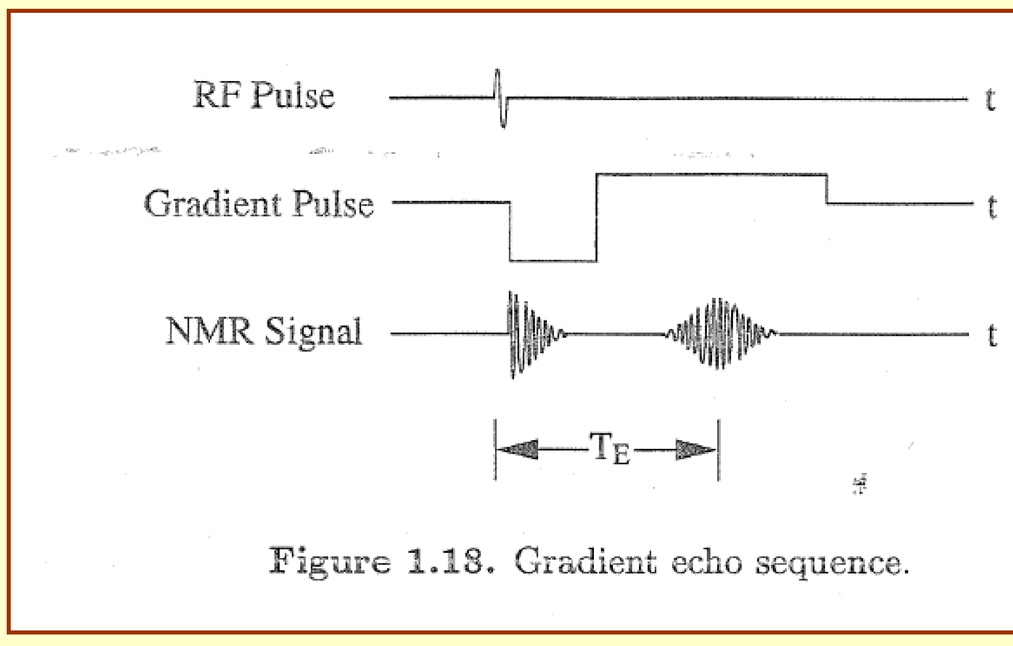
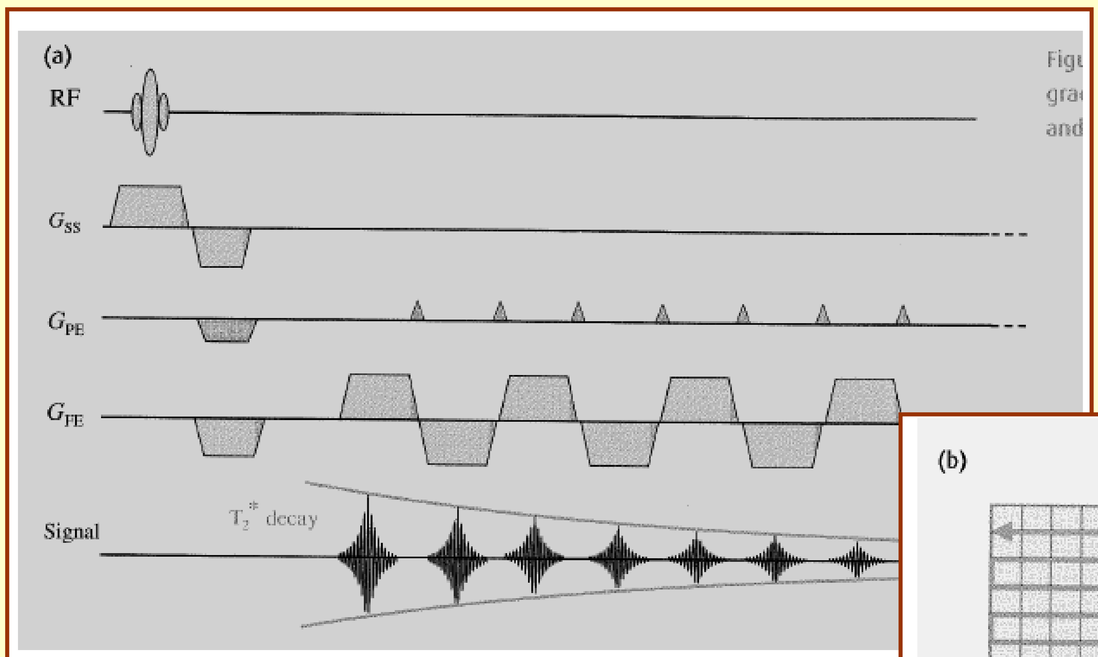
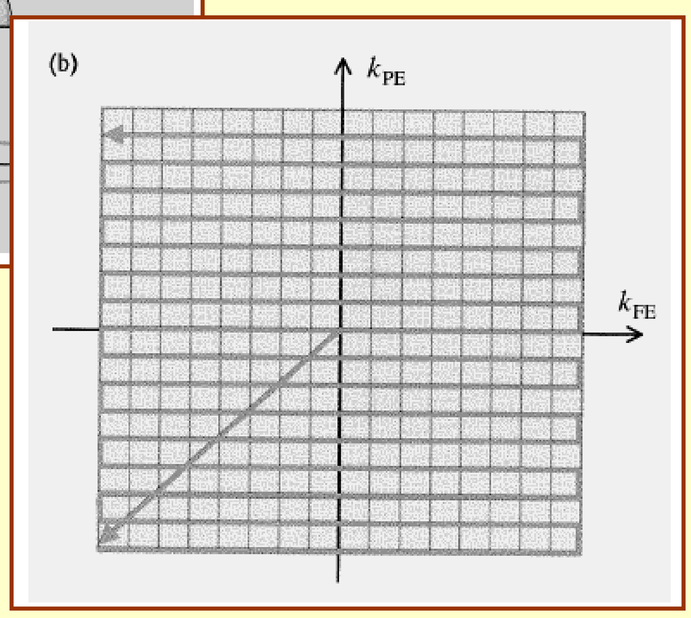


Figure 1.18. Gradient echo sequence.

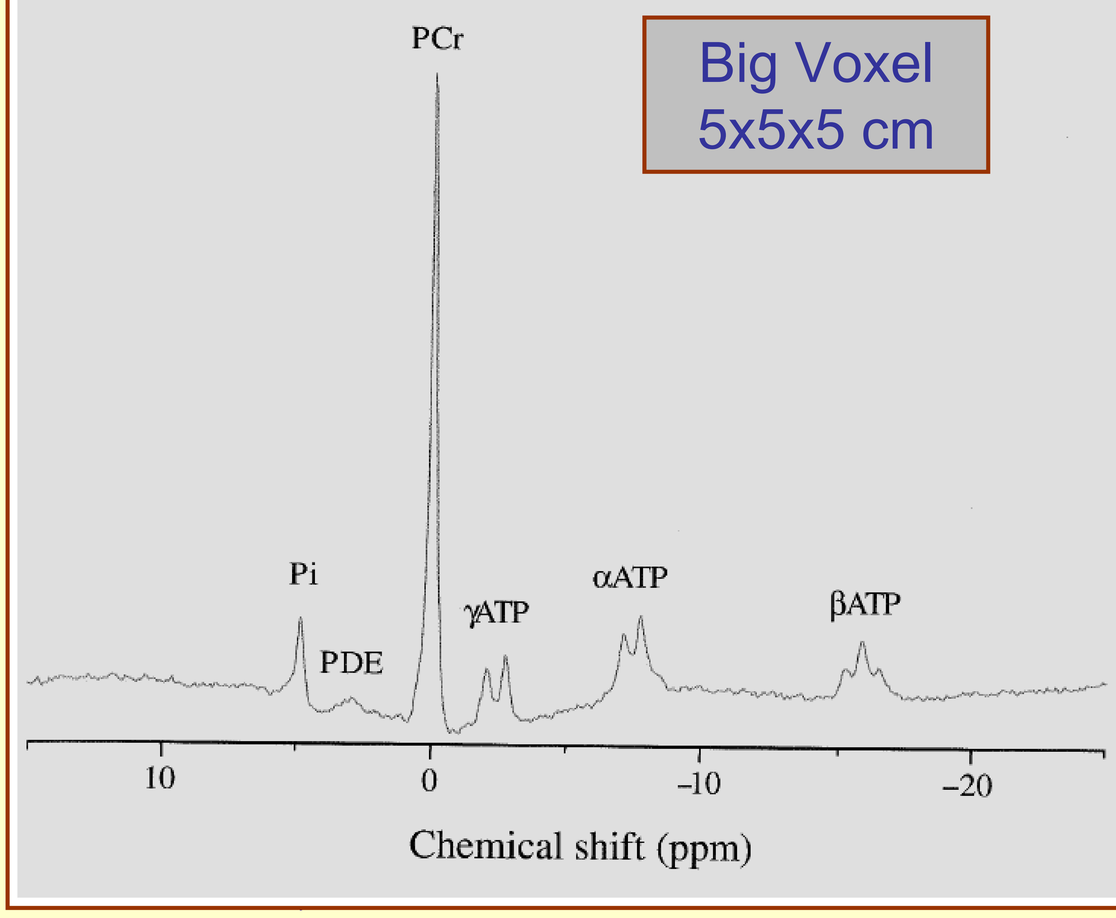
Pulse sequences EPI





Very fast 10 ms/slice, used for fMRI, typically 128x128 voxels

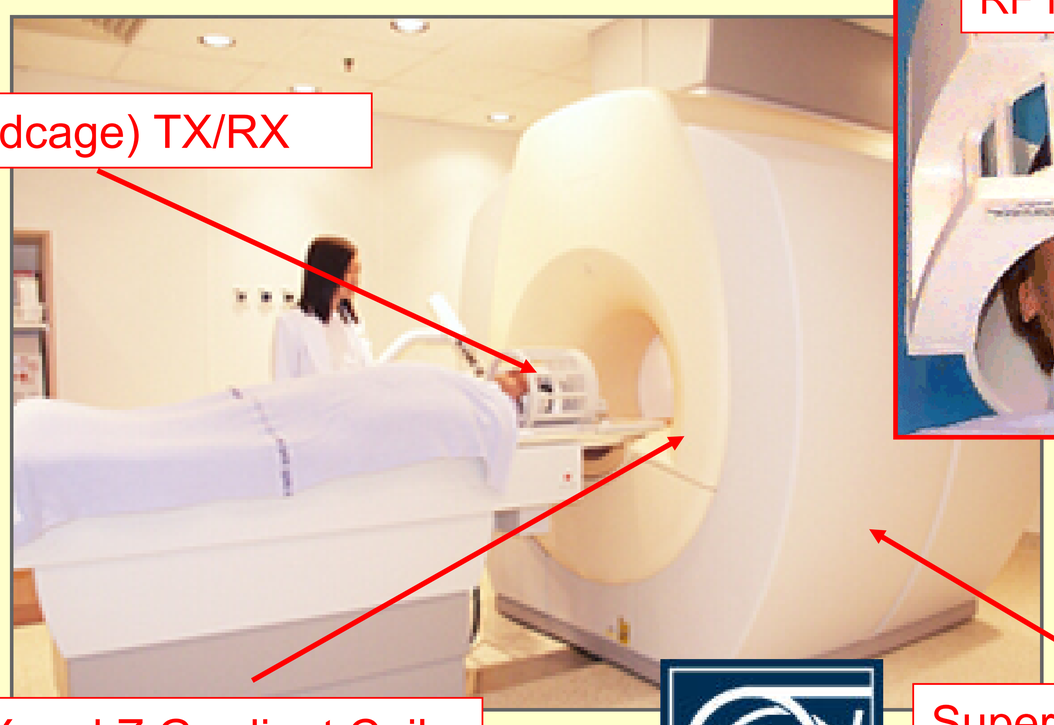
Spectroscopy Possible



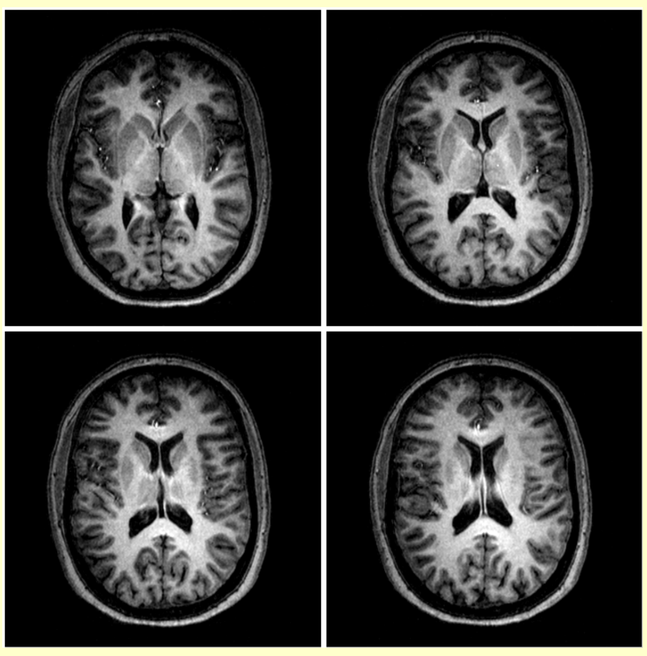
Big Voxel 5x5x5 cm

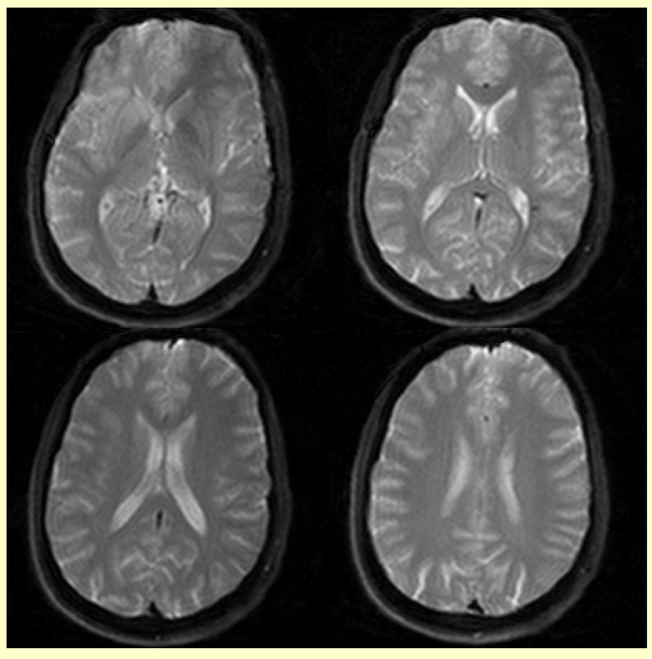
Figure 15.18 Phosphorus spectrum from normal muscle showing the main metabolite peaks. PDE denotes phosphodiesters; Pi inorganic phosphate and PCr phosphocreatine. Courtesy of Dr F. Howe, St George's Hospital Medical School, London.

Hospital MRI Scanner



MRI Soft Tissue Contrast

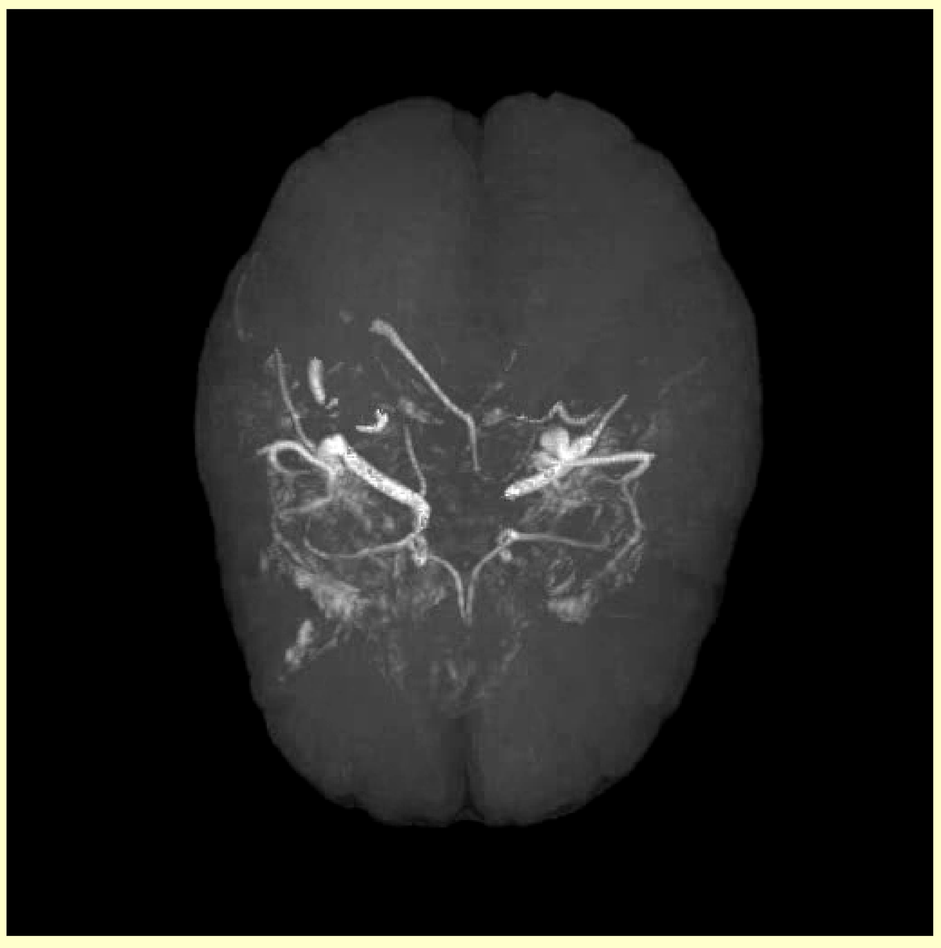




© http://www.wbic.cam.ac.uk

3D Volume Data Sets

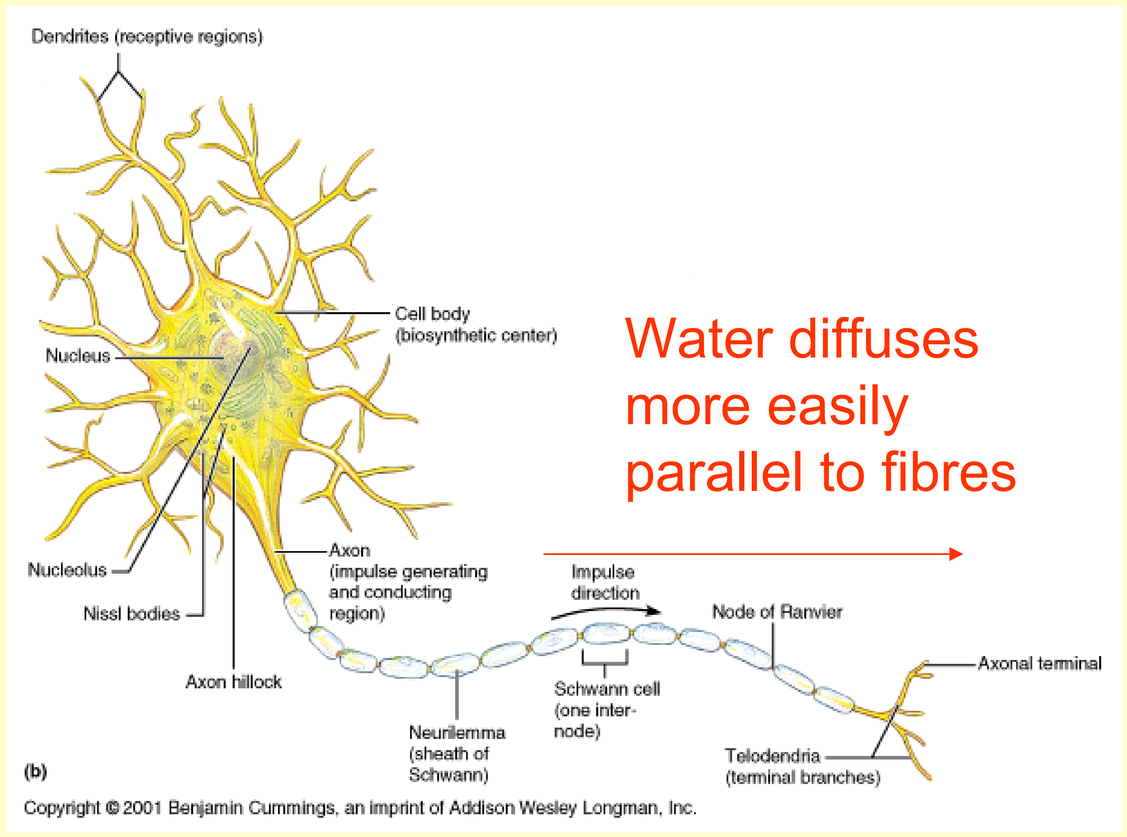
3D Volume Data Sets



Things to do with MRI

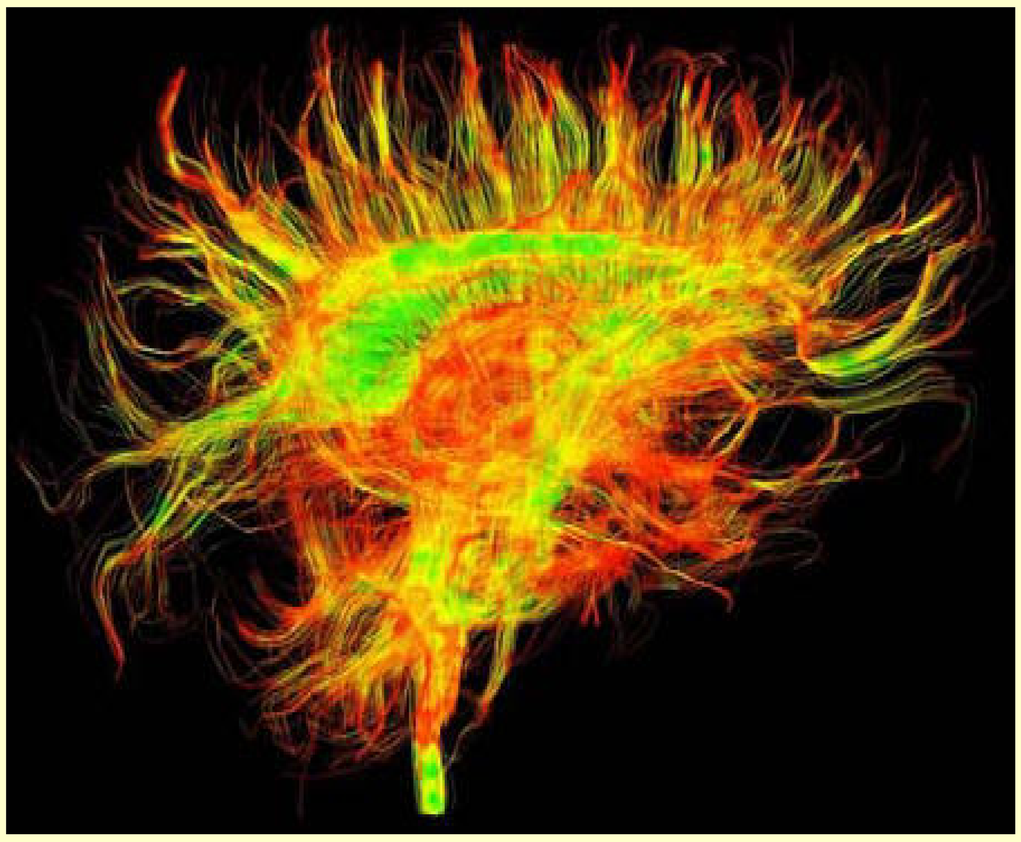
- Anatomy: proton density, T1, T2 etc
- Flow – cardiac
- Diffusion – trace neural pathways
- Functional imaging
  - research
  - surgical planning
- Interventional
- Spectroscopy

Diffusion Imaging

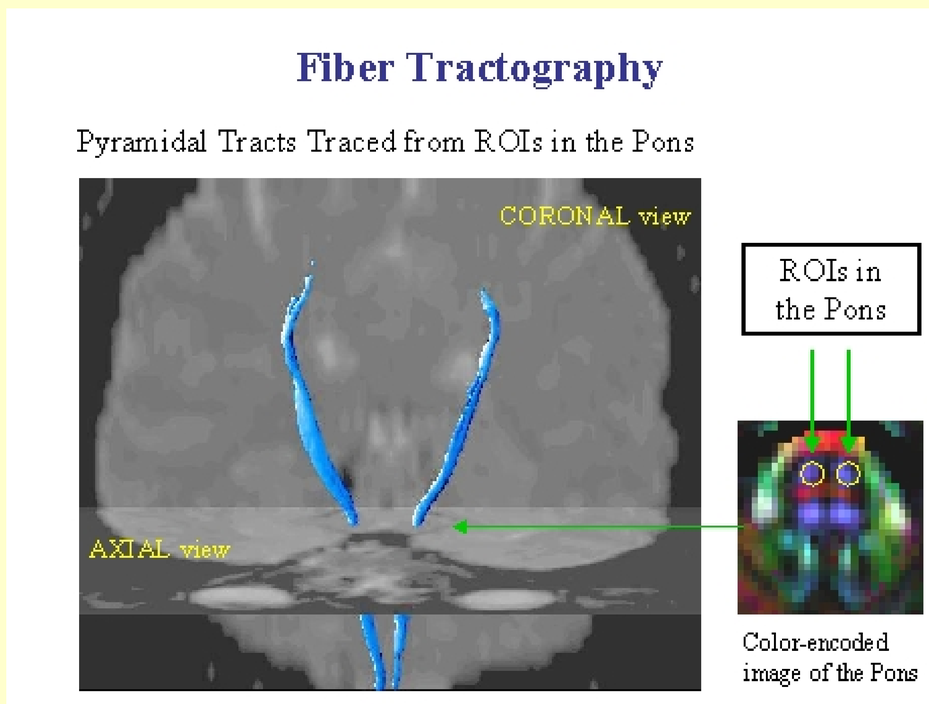


Water diffuses more easily parallel to fibres

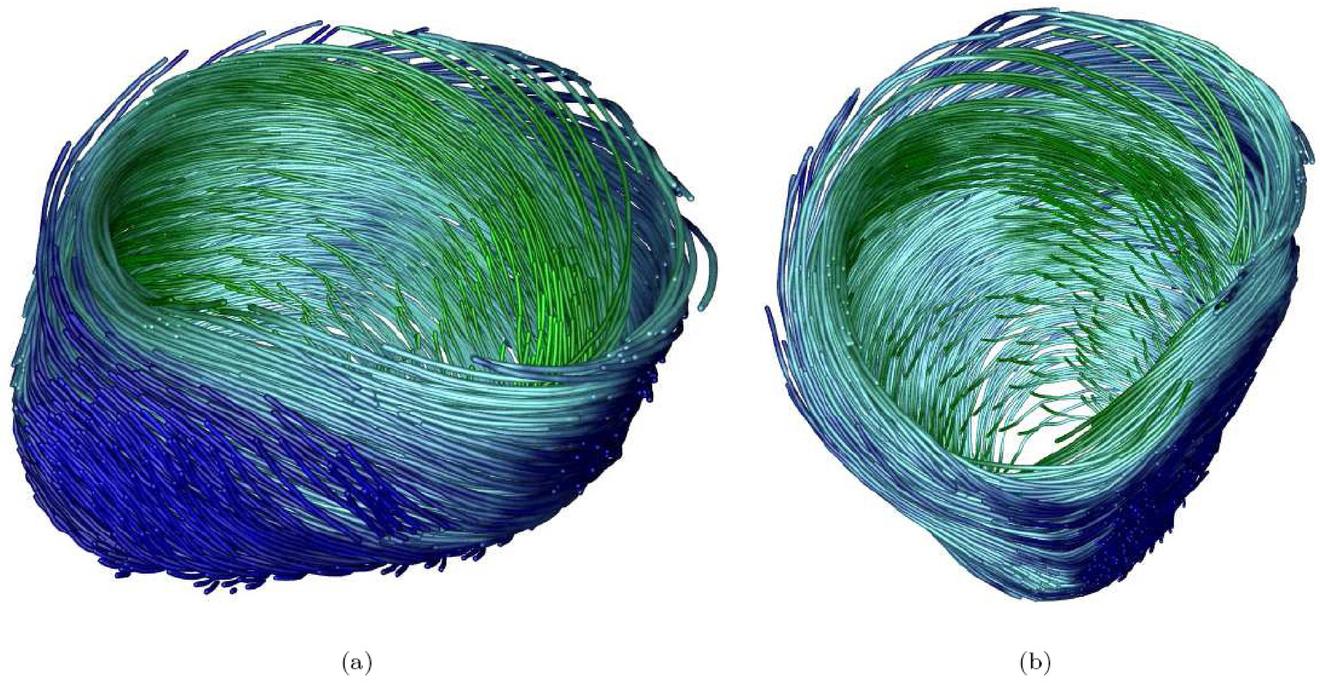
High resolution diffusion image



Diffusion Imaging



Muscle Fibres in Human Heart



(a) (b)

Computed fits to Diffusion MRI of Human Heart

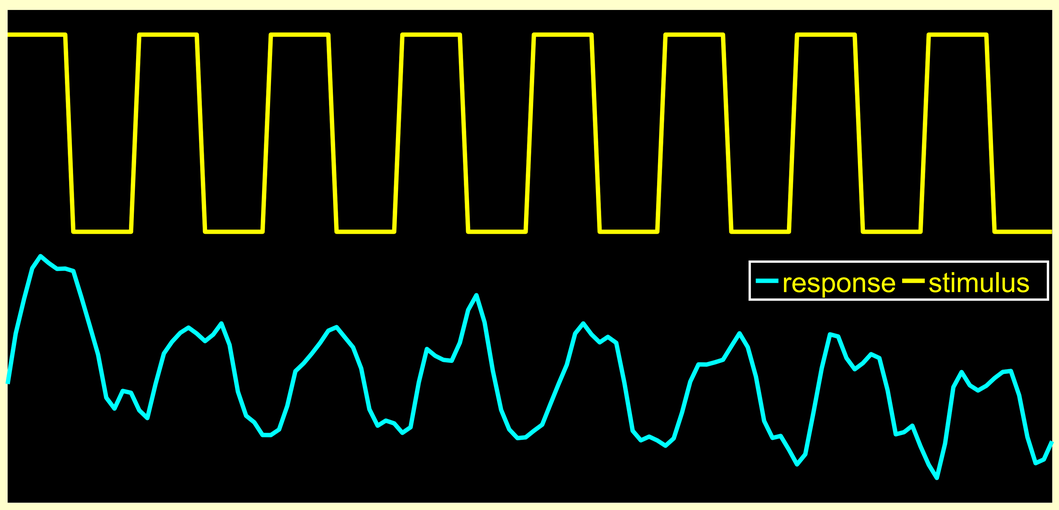
Functional MRI (fMRI)

- Monitor T2 or T2* contrast during cognitive task
- Acquire 20-30 slices every 4 seconds
- Design experiment to have alternating blocks of task and control condition
- Look for statistically significant signal intensity changes correlated with task blocks

Finger Tapping Experiment

Echo-Planar fMRI – Typical Data

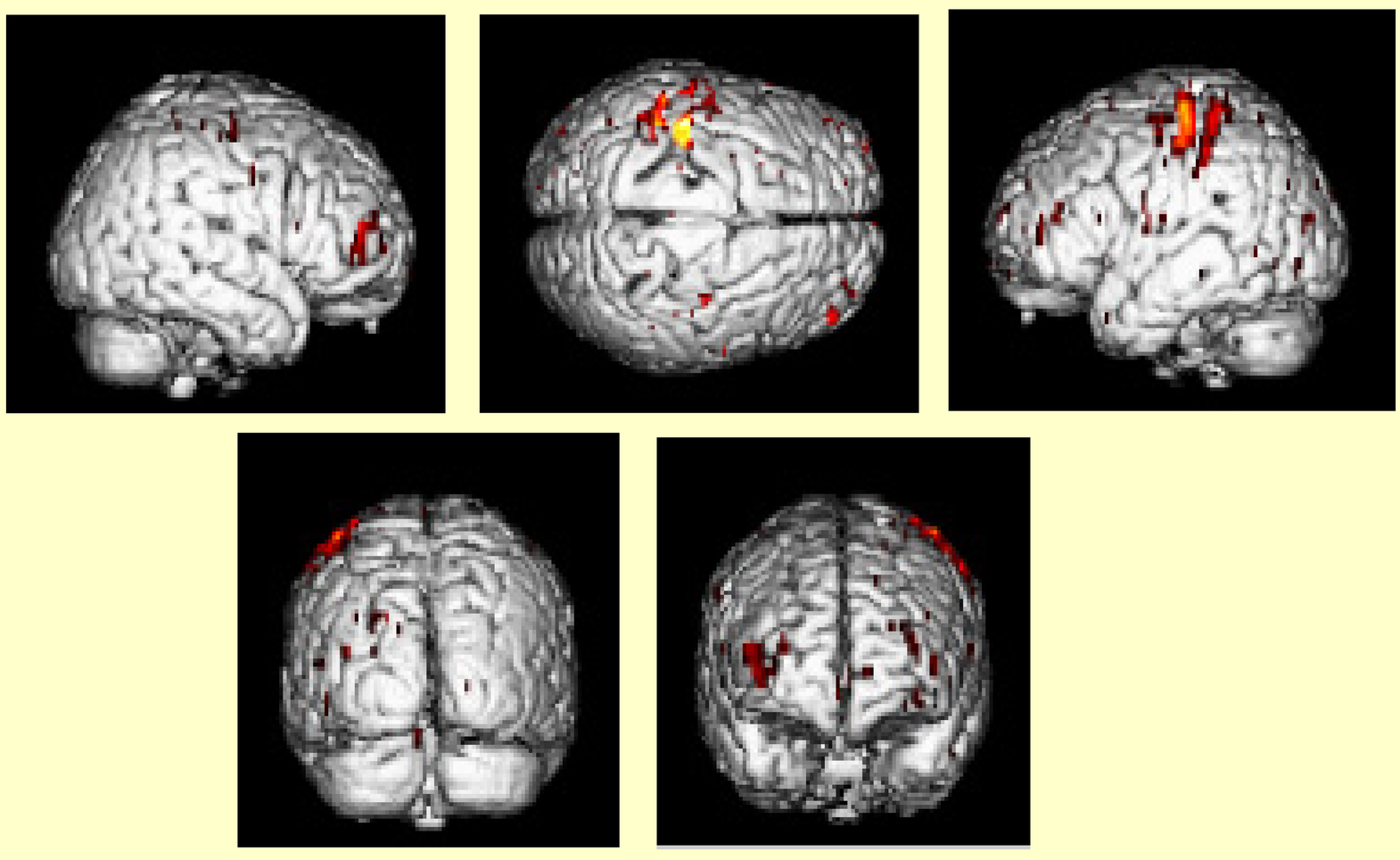
N.B. Signal/Noise ratio is generally poor



GE-EPI images
fMRI correlation maps

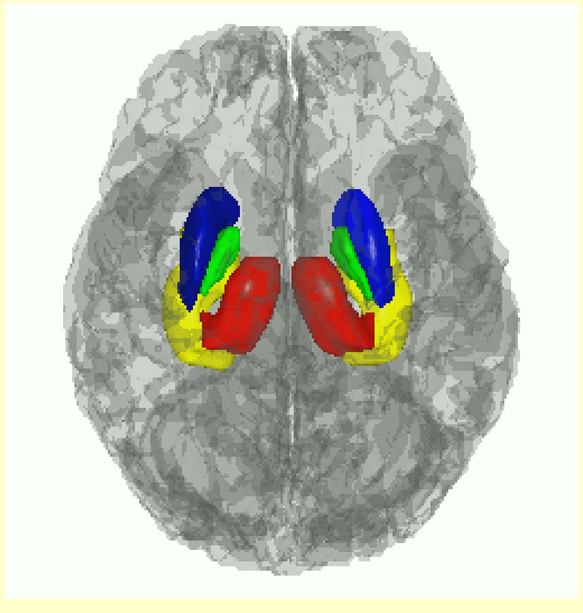
Signal response averaged over region

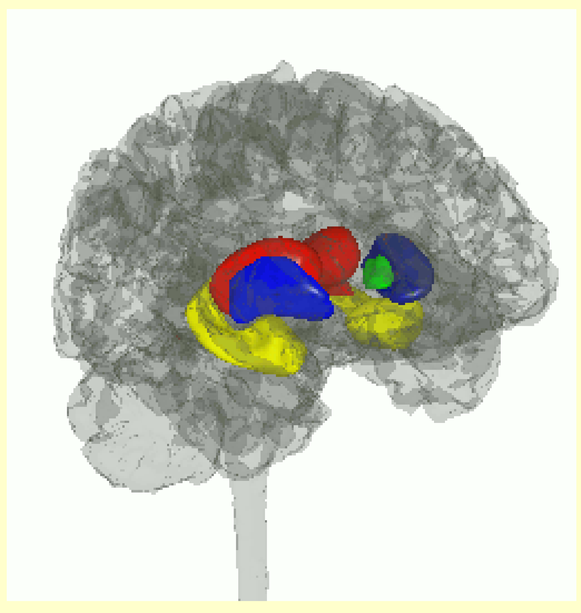
Finger Tapping Experiment



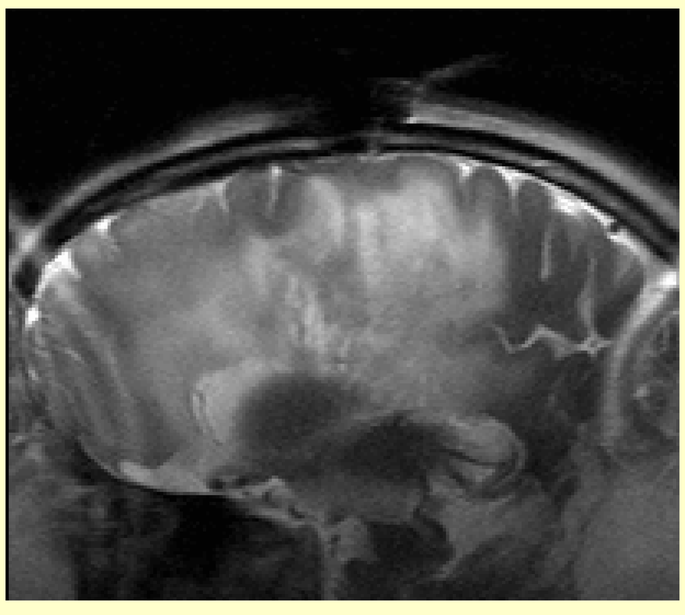
Segmentation

Can we sort out the anatomy?





Interventional MRI

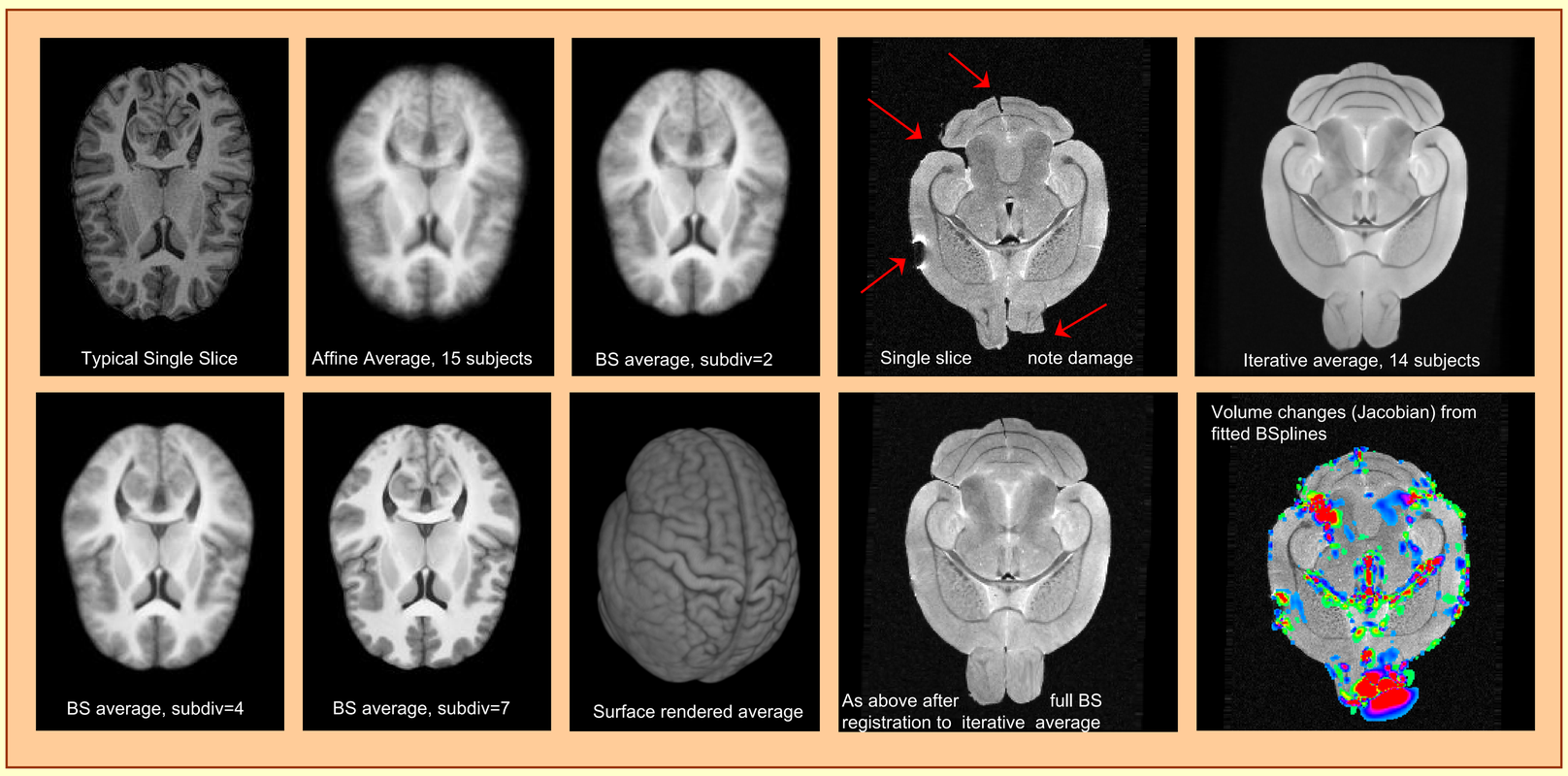


This is still quite rare & expensive

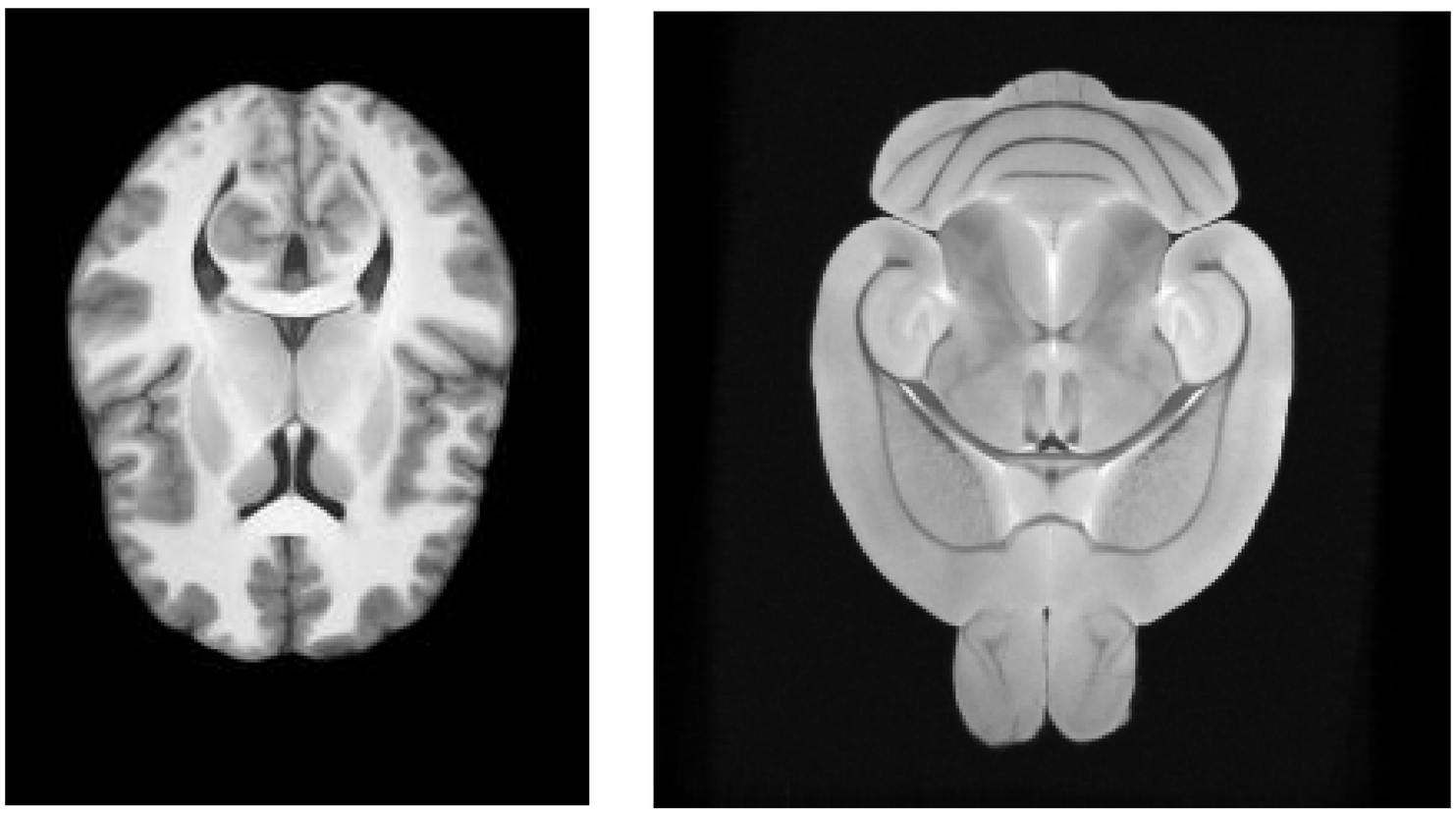
Sensitivity

- 256 x 256 x 256 voxels typical
- Can scale size of RF send/receive coils to size of object.
- Hence typical voxel dimension range is 0.050 mm to several mm.
- BUT signal to noise depends on voxel volume, will never do single cells.
- Overnight acquisitions for mouse brains

Results from MRI: Iterative averages ~15 individuals



Results from MRI: Iterative averages ~15 individuals



Richard Ansorge, IEEE HPMI October 2009

PET

Positron Emission Tomography

Positron Emission Tomography

- Origins in High Energy Physics (CERN).
- Uses Positrons “anti-matter”.
- Positrons annihilate with electrons to make “pure energy”.
- Actually makes 2 photons.

Principle of PET

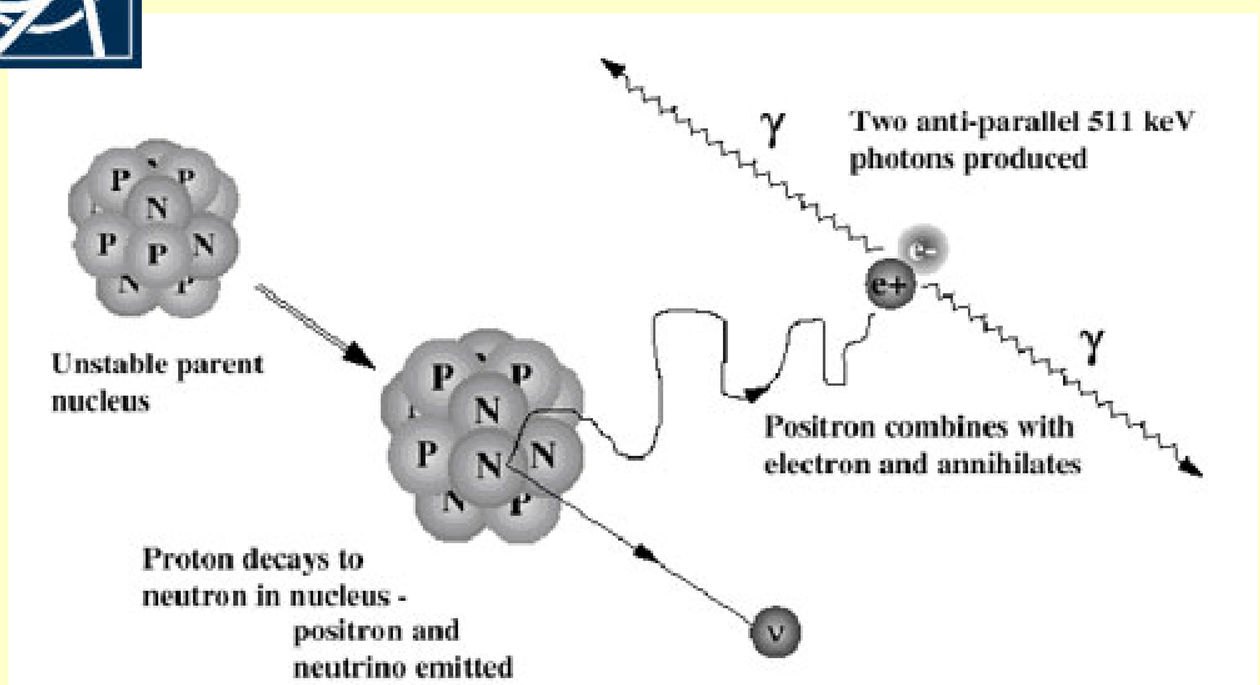
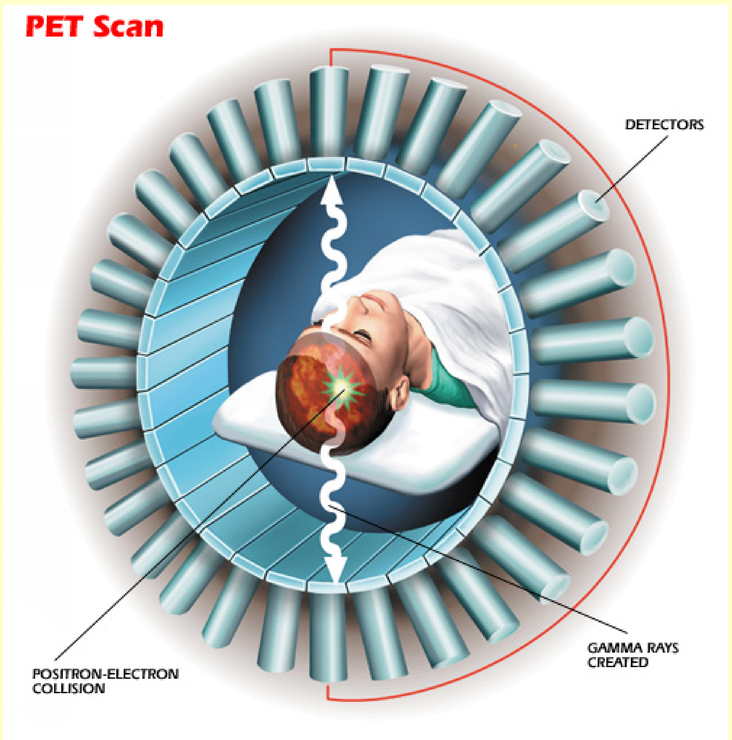


Figure 1.1. Positron emission and annihilation.

Positron Emission Tomography (PET)



Inject (short-lived) positron emitting isotope.

Positron annihilates with electron giving pair of back to back 0.511 MeV gamma rays.

Detect both gammas using fast (5ns) coincidences, get “Line of Response” (LOR). Reconstruction of tracer distribution similar to CT.

Isotopes used in PET

	^{18}F	^{11}C	^{13}N	^{15}O	^{68}Ga
Maximum Energy (MeV)	0.63	0.96	1.20	1.74	1.90
Most Probable Energy (MeV)	0.20	0.33	0.43	0.70	0.78
Half-Life (mins)	110	20.4	9.96	2.07	68.3
Max Range in Water (mm)	2.4	5.0	5.4	8.2	9.1

Addenbrookes Hospital

Tracers Produced

^{15}O (Inhale, H_2O)

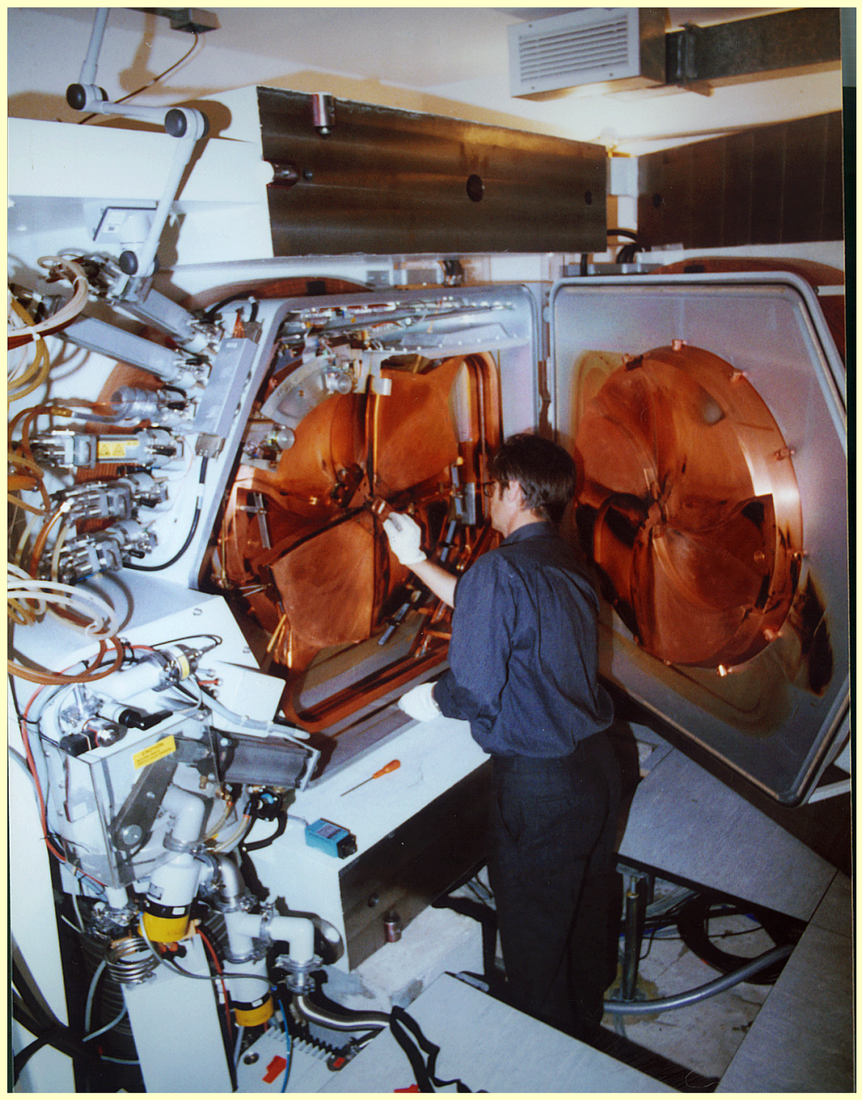
^{11}C (CO, CO_2)

^{18}F (FDG)

…

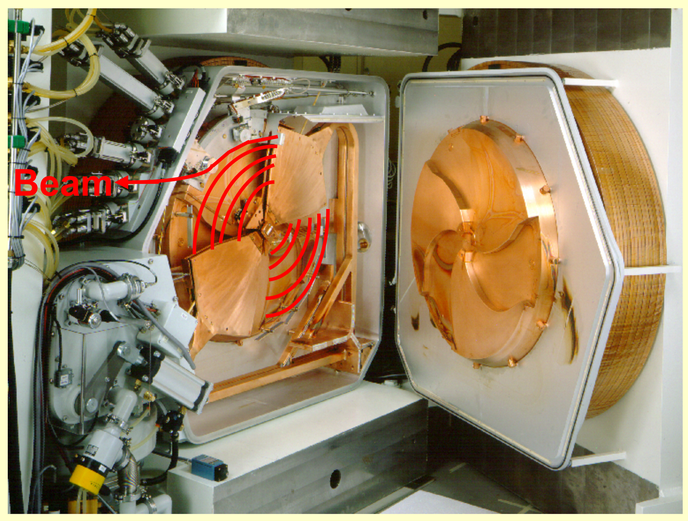
Molecular imaging

GE Medical Systems PETtrace Cyclotron



Addenbrooke's Cyclotron for PET tracers

Beam acceleration

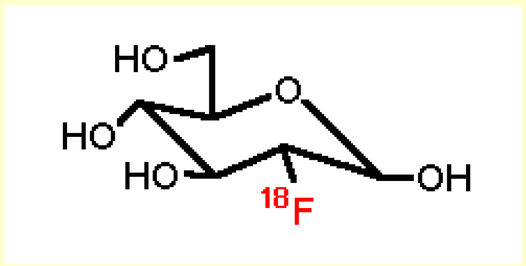


PET is used to image the *function* of specific organs in the body not the anatomy.

PET is *complimentary* to the other imaging modalities.

PET is lower resolution but very *sensitive*.

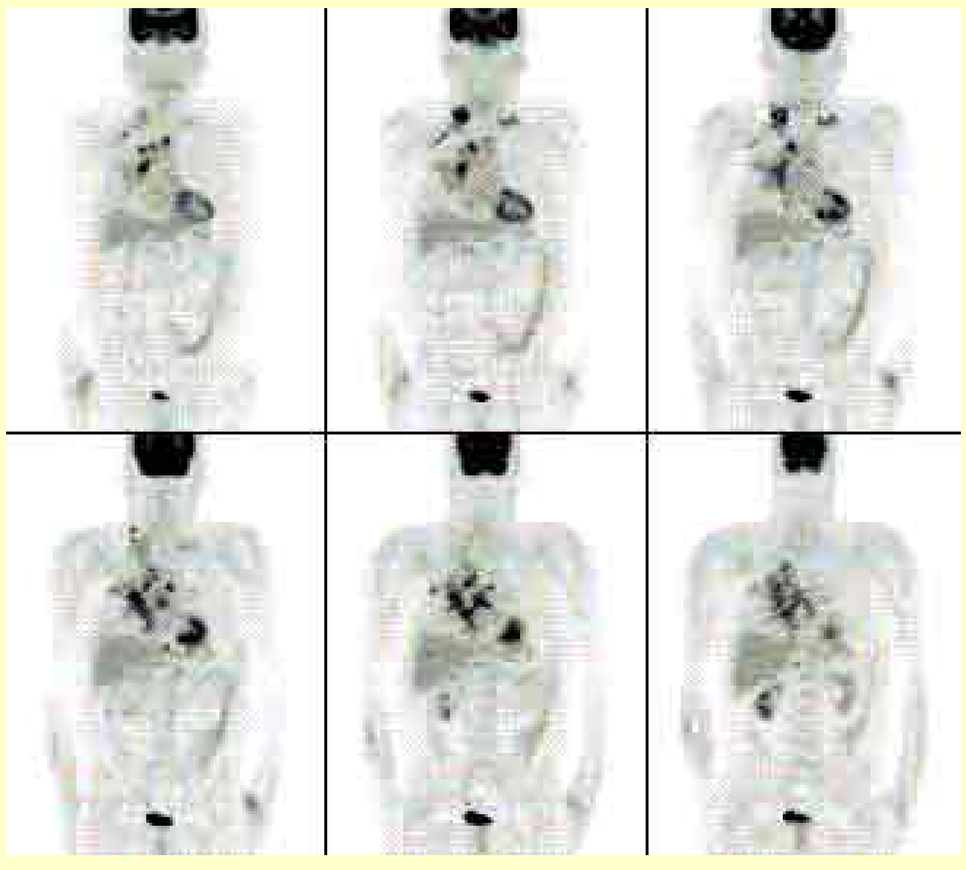
FDG or Fluorodeoxyglucose



The single most important PET tracer.

FDG follows the same metabolic pathway as Glucose, i.e. it is “burnt” in actively metabolizing cells. THEN the ^{18}F stays put. Thus ^{18}F accumulates at “hot-spots” of high metabolic activity.

Whole Body PET

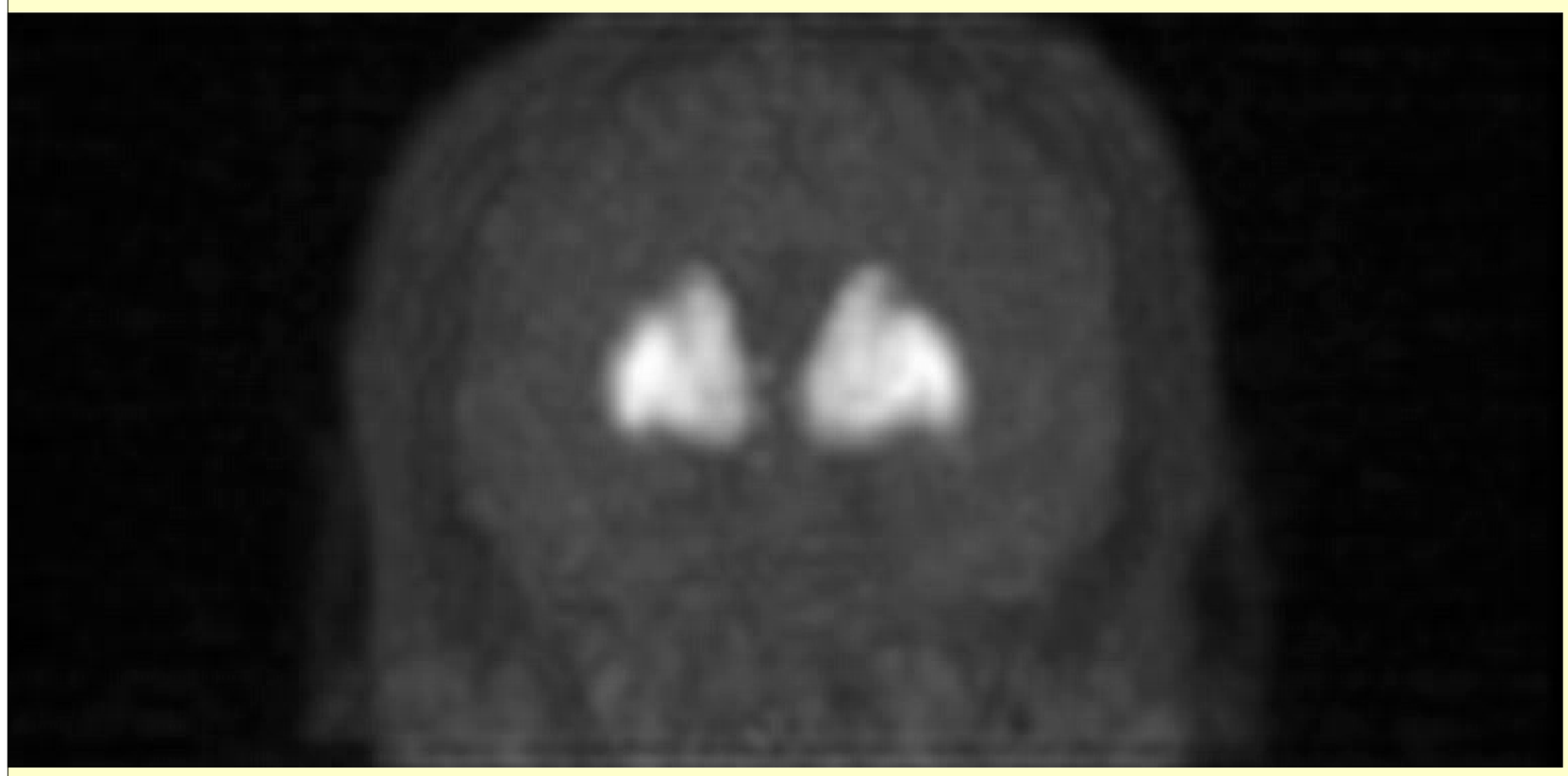


PET Visualization

F-18 fluorodeoxyglucose (FDG). Patient with colorectal cancer. Image is maximum intensity projection through attenuation corrected whole body image, acquired in multiple axial fields-of-view and reconstructed with OSEM algorithm. High uptake is seen in the kidney, liver, bladder, and tumor.

© http://www.cc.nih.gov/pet/images.html

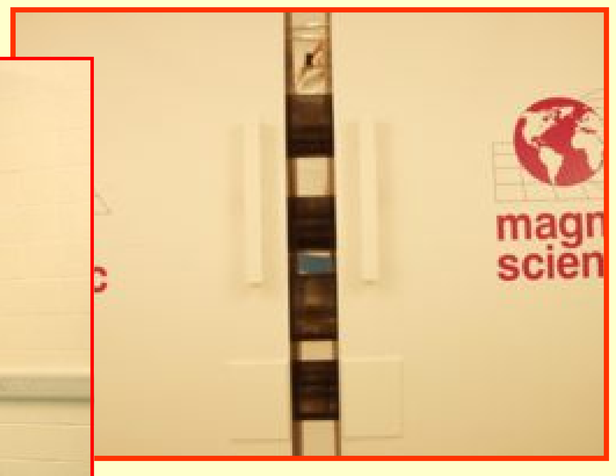
New Tracer: Raclopride in Human Striatum

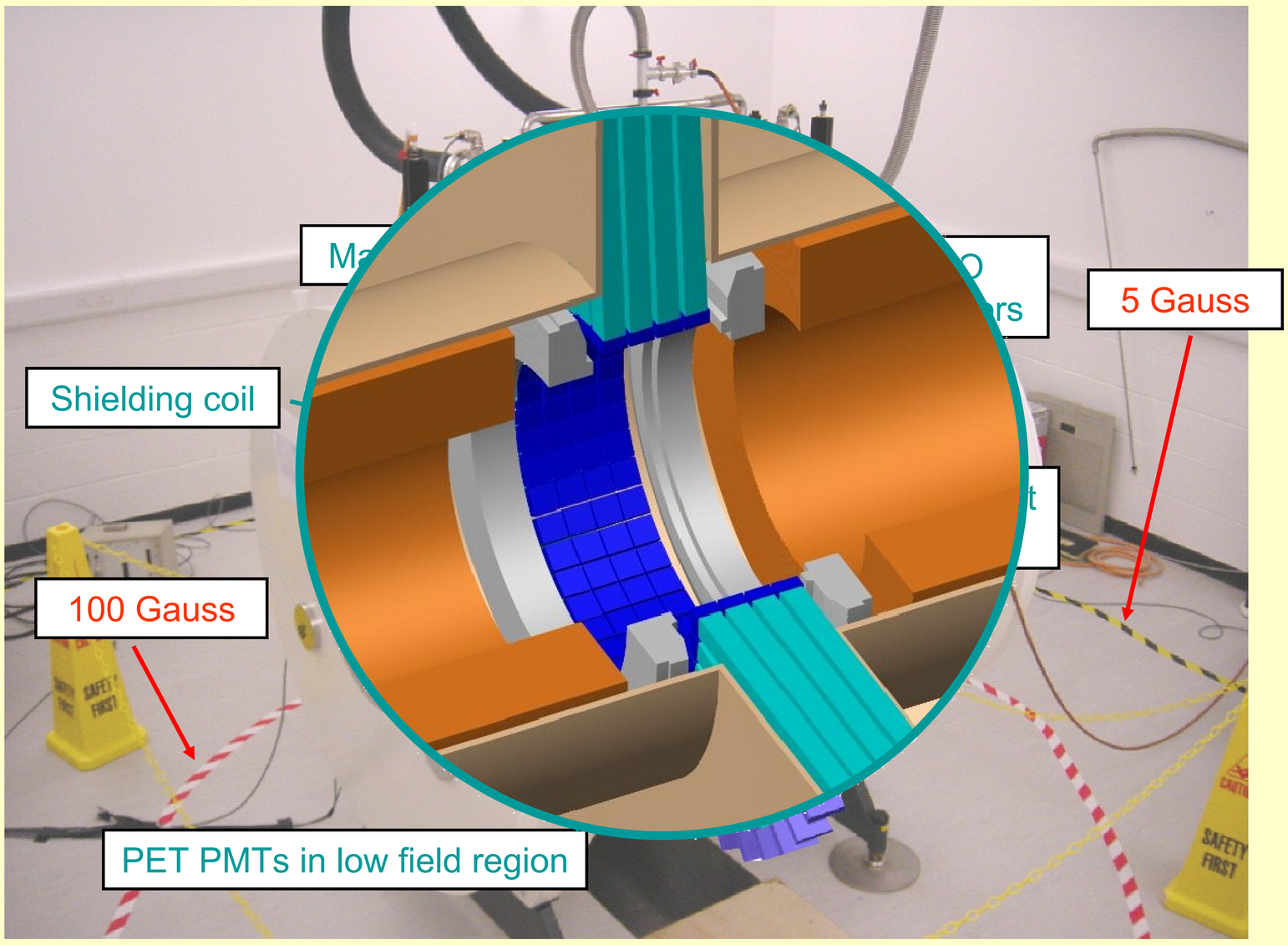


Wolfson Brain Imaging Centre &

MRC Cambridge Centre for Behavioural and Clinical Neuroscience

Combined PET & MRI

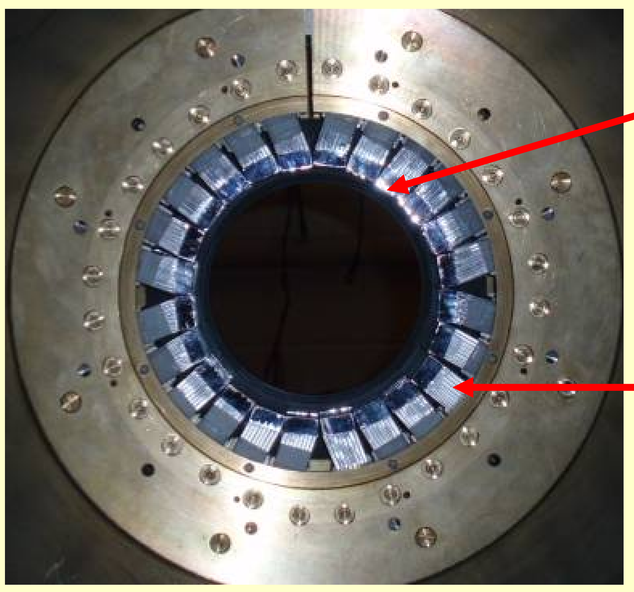




PET detector modules inserted in gap

PMT's housed in magnetic screens and the signal wiring passing through the back plate of the external Faraday cage (cover removed)

View up magnet bore Ring diameter 14.8 cm



LSO crystal blocks in 4 axial rings, 24 blocks per ring

Optical fibre 8x8 bundle

Axial FOV 7.6 cm Radial FOV 10 cm

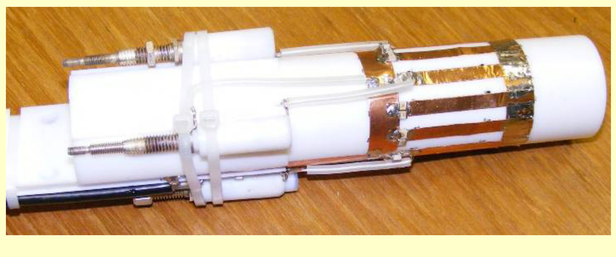
Simultaneous PET & MR images

<u>Derenzo resolution phantom</u>

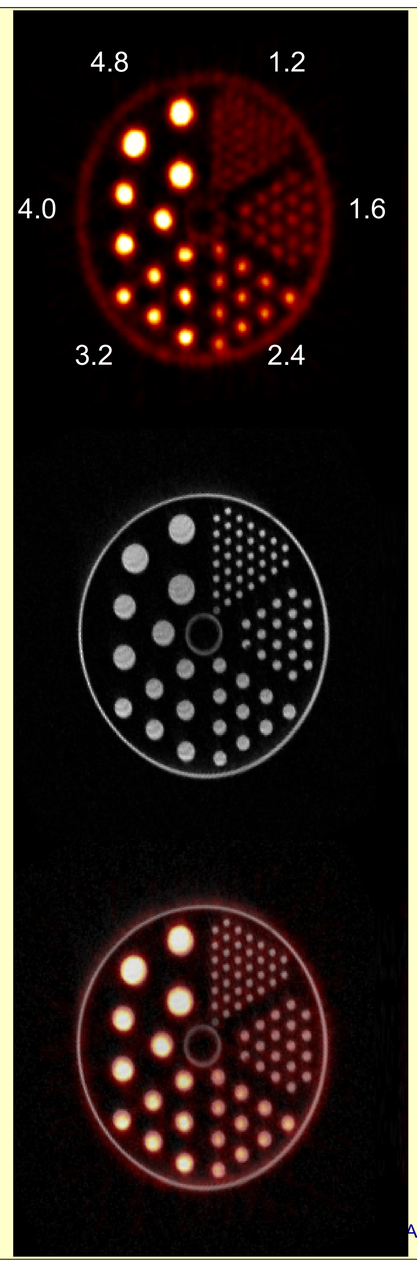
Rod diameters 1.2 mm,
1.6 mm,
2.4 mm,
3.2 mm,
4.0 mm,
4.8 mm

Cylinder diameter 5.0 cm

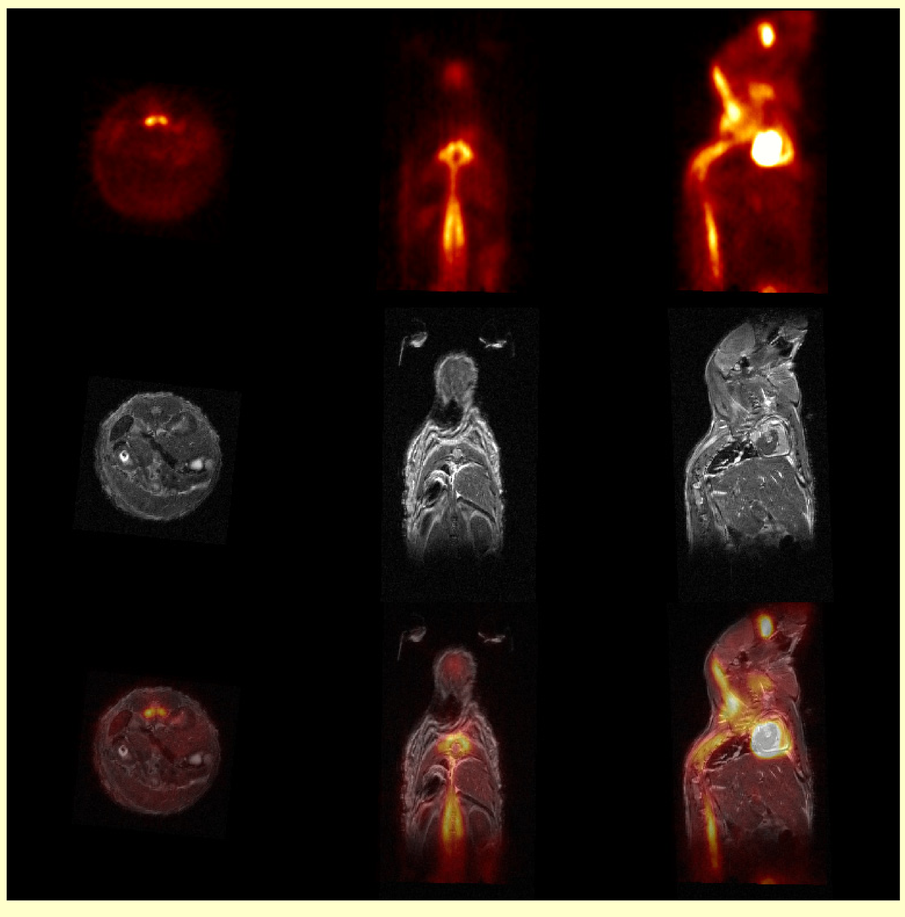
Rod spacing 2 x rod dia



Transmit/Receive birdcage coil



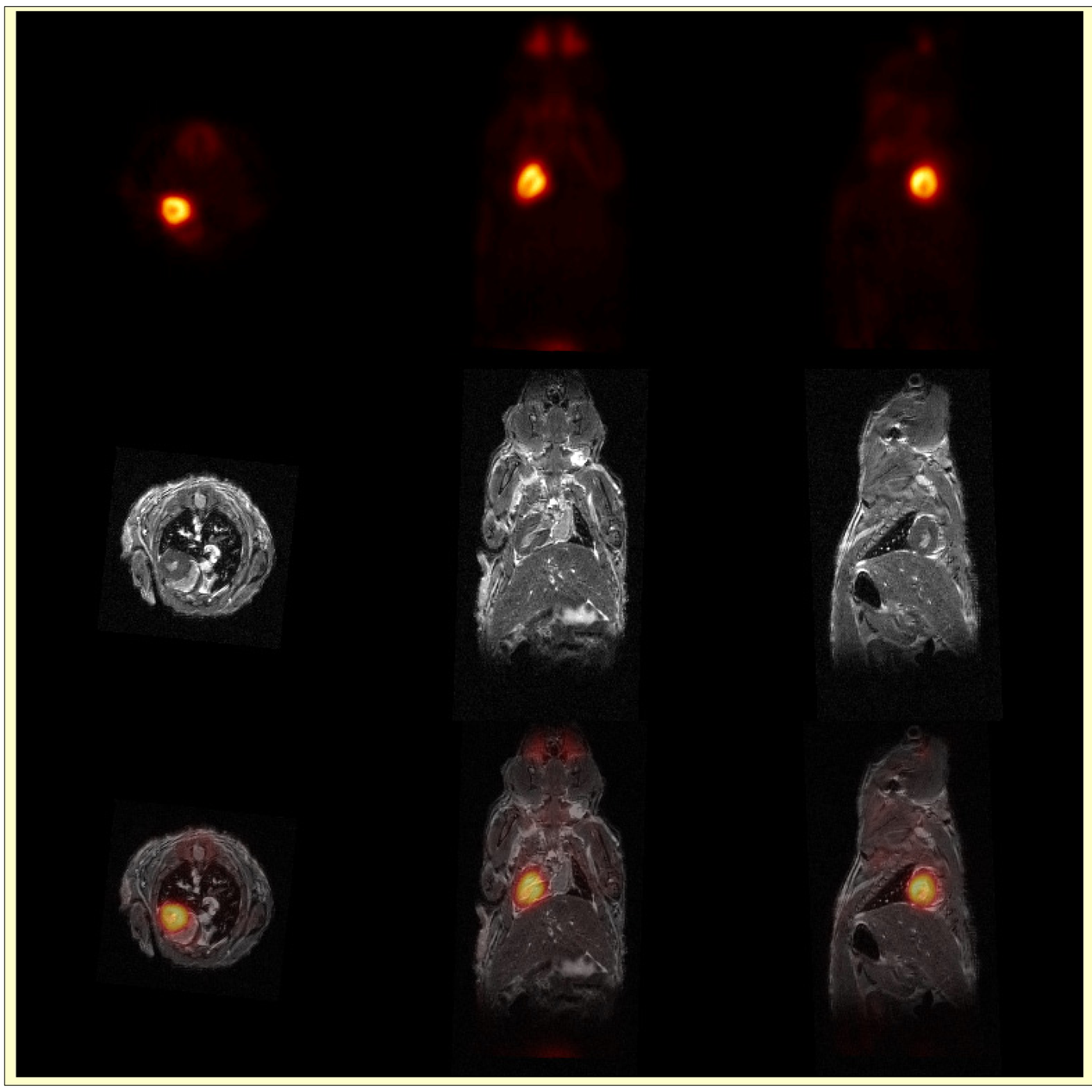
Combined acquisition



An ApoE (genetically modified hyper-lipidemic mouse cadaver)

About four hours post injection of 73.6 MBq of 18F-FDG.

NB this test acquisition was done with a conventional (non-split) gradient coil.



Heart Images

Richard Ansorge

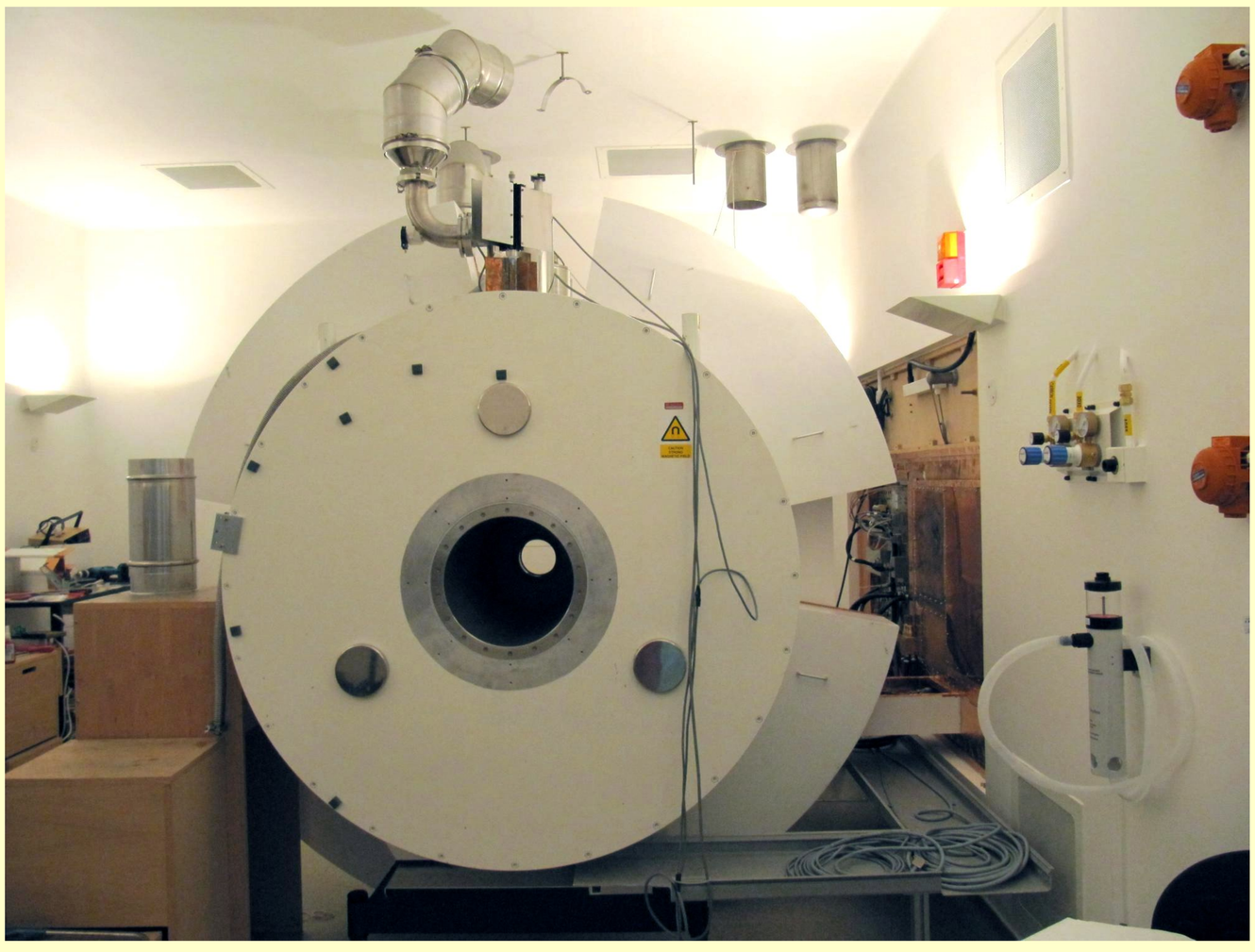
System was moved to the new the Laboratory for Molecular Imaging in March 2008. Expected to be fully operational soon.

Richard Ansorge

Imaging Suite Sept 2009: PET, MRI & PET-MR




09/09/09 Quench Duct Finally Being Installed!



Questions?