Current status of FDPS/ Processor design from HPC perspective

Jun Makino
FS2020 Project and Particle Simulator Research Team, R-CCS
Department of Planetology, Kobe University

R-CCS Cafe, Sep 2, 2019

Talk Overview

- Brief introduction of our team: Co-design team and Particle simulator research team.
- FDPS
  - What don't want to do
  - What to do
  - Current status and future plan
- Processor design from HPC perspective
  - Short summary of the past history of processor architectures for HPC
  - Where are we now and where to go?
- Summary

Team leader, Co-design team, Flagship 2020 project.

Official words: *The mission of the Co-design team is to organize the "co-design" between the hardware and software of the exascale system. It is unpractical to design the many-core complex processor of today without taking into account the requirement of applications. At the same time, it is also unrealistic to develop applications without taking into account the characteristics of the processors on which it will run. The term "Co-design" means we will modify both hardware and software to resolve bottlenecks and achieve best performance.*

In a bit more simpler words...

- Today's microprocessors have become very complex.
- As a result, to develop applications which run efficiently on today's processor has become almost impossible.
- To make the impossible somewhat less impossible, in the early phase of the microprocessor design, predict what problems will occur and fix them if at all possible.

It turned out that there had been really many problems...

Team leader, particle simulator research team.

This one I have been involved since 2012.

Simulation methods for hydrodynamics and structural analysis can be divided into grid-based and particle-based methods. In the latter case, physical values are assigned to particles, while the partial differential equation is approximated by the interactions between particles. Particle-based methods have been used to study phenomena ranging in scale from the molecular to the entire Universe. Historically, software programs for these applications have been developed independently, even though they share many attributes. We are currently developing a" universal " software application that can be applied to problems encompassing molecules to the Universe, and yet runs efficiently on highly parallel computers such as the K computer.

Current Members

- **JM**
- **Daisuke Namekata (R-CCS researchers)**
- **Yutaka Hirai (RIKEN SPD)**
- **Youhei Ishihara (Ph. Student, Kyoto U.)**

Former Member

- Ataru Tanikawa (U. Tokyo)
- Natsuki Hosono (JAMSTEC)
- Takayuki Muranushi
- Steven Rieder (Exeter)
- Long Wang (Bonn)
- Yutaka Maruyama (Architecture Development Team)
- Kentaro Nomura (Kobe U.)
- Keigo Nitadori (Operations and Computer Technologies Division)
- Masaki Iwasawa (Kobe U.)
- David Michael Hernandez (Harvard)

Codesign and Particle Simulator Research teams will be closed by the end of FY 2020 and 2021. So we are trying to find some ways to continue our effort elsewhere. Also, I understand that the total budget of R-CCS research division is far from sufficient.

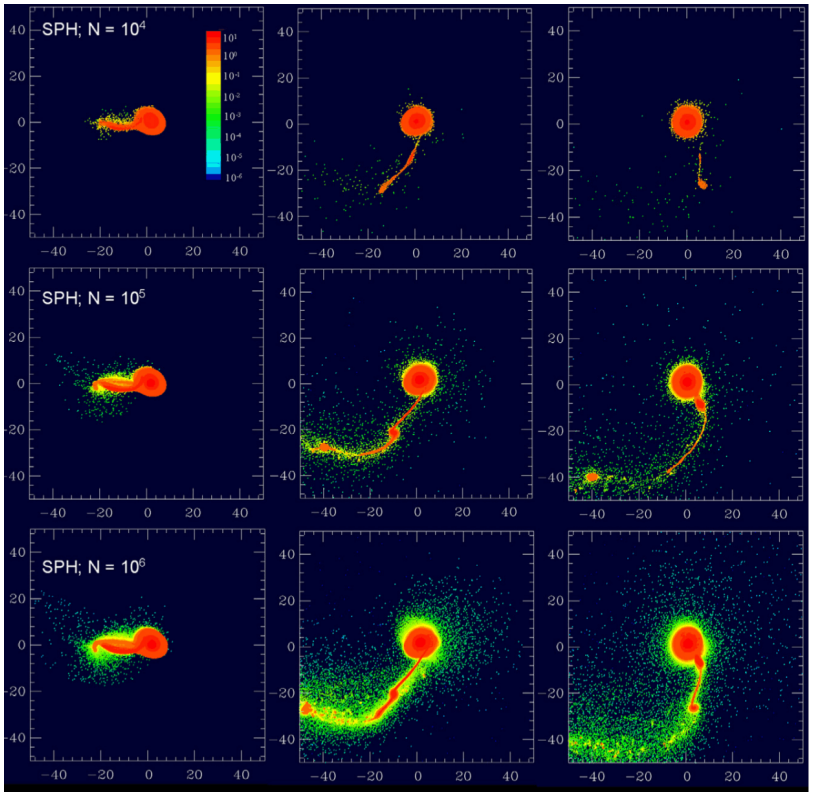
What do we do?

We have developed and maintaining FDPS (Framework for Developing Particle Simulator).

1. **What we (don't) want to do when writing particle-based simulation codes.**
2. **What should be done?**
3. **Design of FDPS**
4. **Current status and future plan**

What we want to do

- We want to try large simulations.
- Computers (or the network of computers...) are fast enough to handle hundreds of millions of particles, for many problems.
- Largest simulations still employ 1M or less particles....



Canup+ 2013

What we want to do

More precisely, what we do not want to do

- **We do not want to write parallel programs using MPI.**
- **We do not want to modify data structure and loop structure to make use of data caches.**
- **We do not want to do complicated optimizations to hide interprocessor communications.**
- **We do not want to write tricky codes to let compilers make use of SIMD instruction sets.**
- **We do not want to do machine-specific optimizations or write codes using machine-specific languages.**

In other words

- Modern high-end HPC systems have become too complex and too difficult to program
  - Wide SIMD instruction set
  - Many-core architecture with hierarchical cache
  - Decreasing memory bandwidth
  - Distributed memory parallel computer without global address space
  - Accelerators

But what we can do?

Traditional ideas

- Hope that parallelizing compiler will solve all problems.
- Hope that big shared memory machine will solve all problems.
- Hope that parallel language (with some help of compilers) will solve all problems.

But...

- These hopes have never been realized.
- Reason: low performance. Only the approach which achieves the best performance on the most inexpensive systems survives.

Then what can we really do?

1. Accept the reality and write MPI programs and do optimization
   Limitation: If you are an ordinary person the achieved performance will be low, and yet it will take more than infinite time to develop and debug programs. Your researcher life is likely to finish before you finish programming. (Also, your target machine will disappear before...)
2. Let someone else do the work
   Limitation: If that someone else is an ordinary person the achieved performance will be low, and yet it will take more than infinite time and money.

- Neither is ideal
- We do need "non-ordinary people".

Problems with "non-ordinary people"

- If you can secure non-ordinary people there might be some hope.
- But they are very limited resource.

If we can apply "non-ordinary people" to many different problems, it could be part of the solution.

How can we apply "non-ordinary people" to many different problems?

Our approach:

- **Formulate an abstract description of the approach of "non-ordinary people", and apply it to many different problem.**
- **"Many different" means particle-based simulations in general (FDPS), or regular-grid calculation (Formura).**
- **Achieve the above by "metaprogramming"**

To be more specific:

Particle-based simulations includes:

- **Gravitational many-body simulations**
- **molecular-dynamics simulations**
- **CFD using particle methods(SPH, MPS, MLS etc)**
- **Meshless methods in structure analysis etc (EFGM etc)**

Almost all calculation cost is spent in the evaluation of interaction between particles and their neighbors (long-range force can be done using tree, FMM, PME etc)

Our solution

Therefore, if we can develop a program which generates a highly optimized MPI program to do

- **domain decomposition (with load balance)**
- **particle migration**
- **interaction calculation (and necessary communication)**

for a given particle-particle interaction, that will be the solution.

Design decisions

- **API defined in C++**
- **Users provide**
  - **Particle data class**
  - **Function to calculate particle-particle interaction**

  Our program generates necessary library functions.
- **Users write their program using these library functions.**

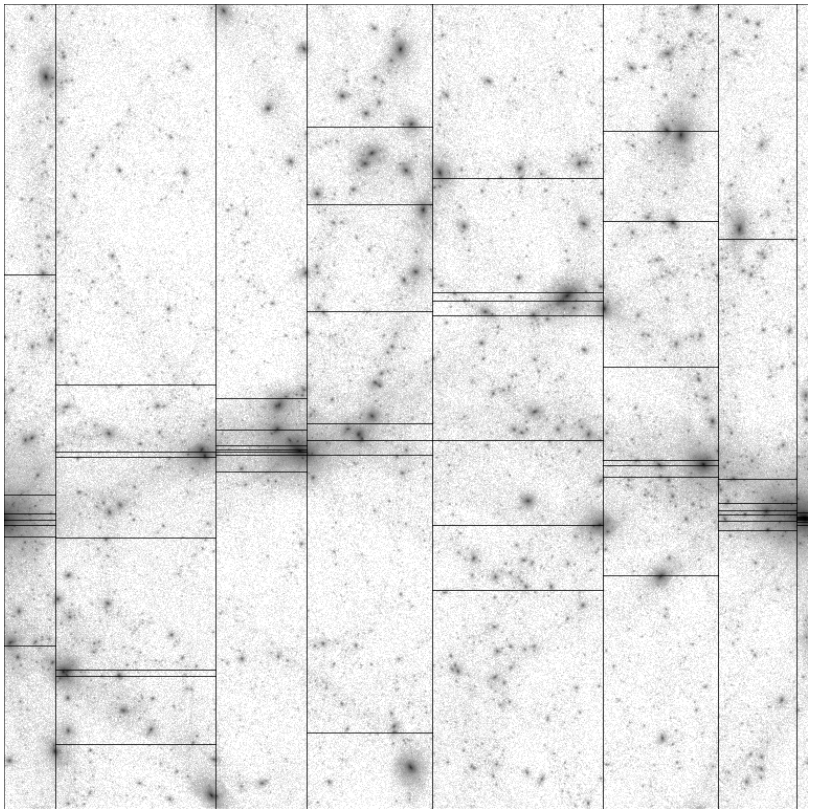
Actual "generation" is done using C++ templates.

Status of the code

- Publicly available (Iwasawa+2016)
- A single user program can be compiled to single-core, OpenMP parallel or MPI parallel programs.
- Parallel efficiency is very high
- Can use GPGPUs or other accelerators efficiently (Iwasawa+ 2019)
- As of version 5.0, the users can use GPUs, and can write their program in Fortran (Namekata+ 2018) or any language with C FFI.

FDPS Github: https://github.com/FDPS/FDPS

Domain decomposition



Each computing node (MPI process) takes care of one domain

Recursive Multisection (JM 2004)

Size of each domain are adjusted so that the calculation time will be balanced (Ishiyama et al. 2009, 2012)

Works reasonably well for up to 160k nodes (so far the max number of processes we could try)

Sample code with FDPS

1. Particle Class

```
#include <particle_simulator.hpp> //required
using namespace PS;
class Nbody{                //arbitorary name
public:
    F64    mass, eps;   //arbitorary name
    F64vec pos, vel, acc; //arbitorary name
    F64vec getPos() const {return pos;} //required
    F64 getCharge() const {return mass;}//required
    void copyFromFP(const Nbody &in){   //required
        mass = in.mass;
        pos  = in.pos;
        eps  = in.eps;
    }
    void copyFromForce(const Nbody &out) { //required
        acc = out.acc;
    }
```

Particle class (2)

```
    void clear() { //required
        acc = 0.0;
    }
    void readAscii(FILE *fp) {//to use FDPS IO
        fscanf(fp,
               "%lf%lf%lf%lf%lf%lf%lf%lf",
               &mass, &eps, &pos.x, &pos.y, &pos.z,
               &vel.x, &vel.y, &vel.z);
    }
    void predict(F64 dt) { //used in user code
        vel += (0.5 * dt) * acc;
        pos += dt * vel;
    }
    void correct(F64 dt) { //used in user code
        vel += (0.5 * dt) * acc;
    }
};
```

Interaction function

```
template <class TParticleJ>
void CalcGravity(const FPGrav * ep_i,
                 const PS::S32 n_ip,
                 const TParticleJ * ep_j,
                 const PS::S32 n_jp,
                 FPGrav * force) {
    PS::F64 eps2 = FPGrav::eps * FPGrav::eps;
    for(PS::S32 i = 0; i < n_ip; i++){
        PS::F64vec xi = ep_i[i].getPos();
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
```

Interaction function

```
        for(PS::S32 j = 0; j < n_jp; j++){
            PS::F64vec rij    = xi - ep_j[j].getPos();
            PS::F64    r3_inv = rij * rij + eps2;
            PS::F64    r_inv  = 1.0/sqrt(r3_inv);
            r3_inv  = r_inv * r_inv;
            r_inv  *= ep_j[j].getCharge();
            r3_inv *= r_inv;
            ai     -= r3_inv * rij;
            poti   -= r_inv;
        }
        force[i].acc += ai;
        force[i].pot += poti;
    }
}
```

Time integration (user code)

```
template<class Tpsys>
void predict(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].predict(dt);
}

template<class Tpsys>
void correct(Tpsys &p,
             const F64 dt) {
    S32 n = p.getNumberOfParticleLocal();
    for(S32 i = 0; i < n; i++)
        p[i].correct(dt);
}
```

Calling interaction function through FDPS

```
template <class TDI, class TPS, class TTFF>
void calcGravAllAndWriteBack(TDI &dinfo,
                             TPS &ptcl,
                             TTFF &tree) {
    dinfo.decomposeDomainAll(ptcl);
    ptcl.exchangeParticle(dinfo);
    tree.calcForceAllAndWriteBack
        (CalcGrav<Nbody>(),
         CalcGrav<SPJMonopole>(),
         ptcl, dinfo);
}
```

Main function

```
int main(int argc, char *argv[]) {
    F32 time  = 0.0;
    const F32 tend  = 10.0;
    const F32 dtime = 1.0 / 128.0;
    // FDPS initialization
    PS::Initialize(argc, argv);
    PS::DomainInfo dinfo;
    dinfo.initialize();
    PS::ParticleSystem<Nbody> ptcl;
    ptcl.initialize();
    // pass initeraction function to FDPS
    PS::TreeForForceLong<Nbody, Nbody,
        Nbody>::Monopole grav;
    grav.initialize(0);
    // read snapshot
    ptcl.readParticleAscii(argv[1]);
```

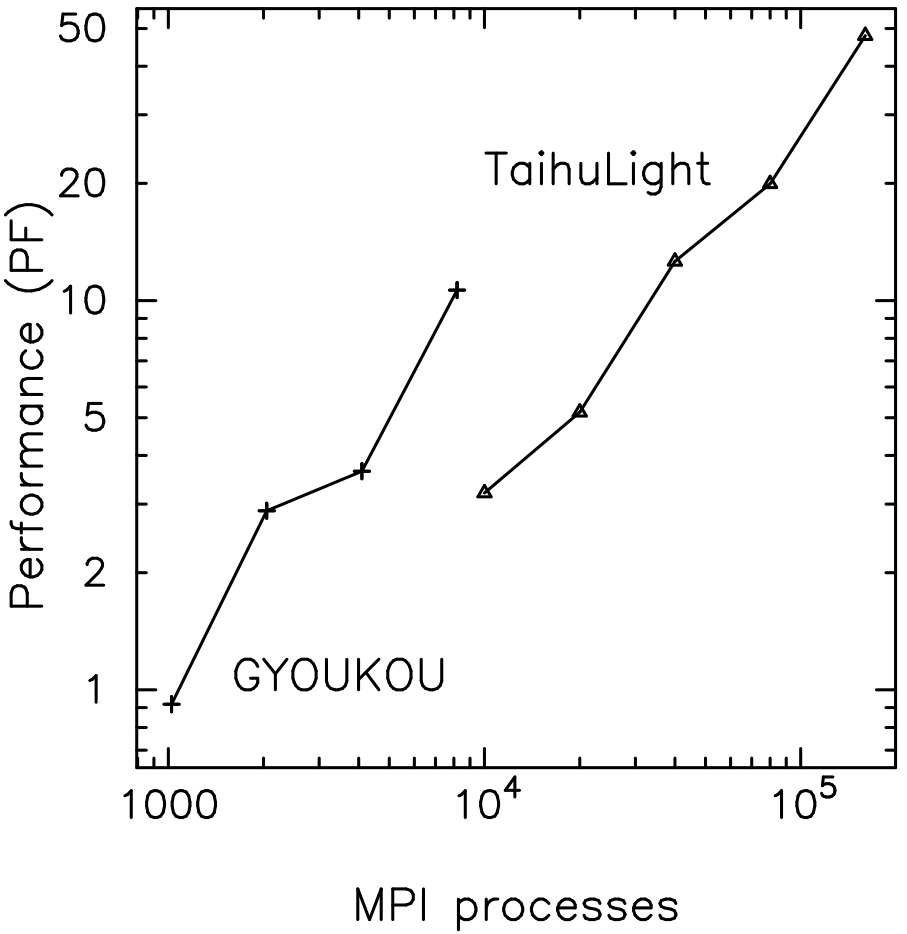
Main function

```
    // interaction calculation
    calcGravAllAndWriteBack(dinfo,
                            ptcl,
                            grav);
    while(time < tend) {
        predict(ptcl, dtime);
        calcGravAllAndWriteBack(dinfo,
                                ptcl,
                                grav);
        correct(ptcl, dtime);
        time += dtime;
    }
    PS::Finalize();
    return 0;
}
```

Remarks

- User-defined particle class should have several "required" member functions
- Multiple particles can be defined (such as dark matter + gas)
- User-defined interaction function should be optimized to a given architecture for the best performance (for now)
- This program runs fully parallelized with OpenMP + MPI(taken care within FDPS)
- SIMD should be taken care in interaction function. Accelerator should be too.

Performance example



- **Weak scaling with 10M particles/process**
- **Simulation of planetary rings**
- **Optimized version for PEZY-SC2 and Sunway TaihuLight**
- **40% and 23.5% of the theoretical peak performance.**

Users of FDPS

- So far, $>$ 30 scientific papers have been published
- Astrophysics, Planetary Science, Material Science, and more
- There are many more users in various fields

FDPS summary

Iwasawa+2016 (Publ. Astron. Soc. J. 2016, 68, 54/arXive 1601.03138, https://github.com/FDPS/FDPS)

- **FDPS offers library functions for domain decomposition, particle exchange, interaction calculation using tree.**
- **Can be used to implement pure Nbody, SPH, or any particle simulations with two-body interactions.**
- **Use essentially the same algorithm as used in our treecode implementation on K computer (GreeM, Ishiyama, Nitadori and JM 2012).**
- **Runs efficiently on K, Xeon clusters or GPU clusters**

What we learned from FDPS development

- Frameworks like FDPS can be used to make good use of a wide variety of processor architectures
- "easy to use" does not necessarily mean "easy to achieve high efficiency"
- "High efficiency" does not necessarily mean high energy efficiency

The goal of HPC R&D: Provide tools to solve scientific problems "efficiently"

Question:

What architecture can provide the "best" performance?

How we can make the R&D of HPC software and hardware "Scientific"?

What do I mean by scientific?

- **Our approach for HPC application development is rather "problem-driven". Try an existing code on a new architecture, see what happens, and fix the problems,**
- **Our approach for HPC architecture is, well, "evolutionary" at best.**
- **Scientific approach should be driven by the first principle, whatever it is.**

Short summary of the past history of processor architectures for HPC

1. CDC 6600/7600
2. Cray-1
3. MIPS
4. IBM Power
5. Intel Pentium 4
6. AMD Athlon 64 X2
7. Intel Xeon Phi

CDC 6600/7600 (1964)

- The first machine to use register scoreboarding = the first machine with efficient superscalar out-of-order (OOO) execution
- (Probably) the first machine to have a large number (16-32) of general-purpose registers

Register Scoreboarding

For an instruction sequence like

```
1  R5 = R1+R2
2  R6 = R3+R4
3  R7 = R5+R6
```

- **Instructions 1 and 2 can be issued in parallel, while Instruction 3 should wait the completion of previous two instructions. Register scoreboarding tells when instruction 3 can be started.**
- **A different approach is to let the compiler schedule instructions**
- **For binary compatibility, OOO is necessary.**

Cray-1 (1976)

- Successor of CDC-7600 (or canceled 8600)
- Vector instruction set and vector registers.
- SRAM memory made it possible to achieved B/F=4 (B/F of XMP was 12)
- Successors: XMP, YMP, C- and T-90: Physical memory shared by multiple processors. B/F kept very high.
- To keep high B/F had become impossible. B/F of NEC vector machines have reduced to 0.5 with SC-Aurora

MIPS (1985)

- Original MIPS = Microprocessor without Interlocked Pipeline Stages
- Thus, all instructions have the same latency and instruction scheduling was done by software
- In-order scalar processor
- OOO execution was adopted finally with R10K (1996)

IBM Power1 (1990)

First microprocessor with register renaming
For an instruction sequence like

```
1  R1 = M1
2  R2 =R1*R1
3  M1 = R2
4  R1 = M2
5  R2 =R1*R1
6  M2 = R2
   ...
```

(something like `for(i=0;i<n;i++)a[i]*=a[i];`**)**
You can issue instruction 4 just after instruction 1,if "R1" of instruction 4 is physically different from that of instruction 1. Register renaming takes care of this mapping between "architecture" registers and "physical" registers

Intel Pentium 4 (2000)

- **First microprocessor with SIMD instruction for double 64-bit words. (SSE2).**
- **SSE2 evolved eventually to AVX-512.**
- **Intel Xeon processors of 2019 have up to two AVX-512 units per core. Thus, they can perform 16 double-precision FMA operations per cycle.**

AMD Athlon 64 X2 (2005)

- First high-performance multi-core microprocessor
- Now we have up to 64 cores/package with AMD EPYC 7002
- Large number of cores means multiple levels of cache memory.

Intel Xeon Phi (2012-2017)

- **Intel's many-core processor for HPC.**
- **AVX-512, 60-70 cores/die (package)**
- **1st gen Knights Corner: 1 AVX512 unit/core**
- **2nd gen Knights Landing: 2 AVX512 units/core**
- **3rd gen Knights Hills: project canceled**

Summary of the history

- Modern processors used for HPC share the following features:
  superscalar, OOO, register renaming, SIMD instruction set, deep cache hierarchy
- In the last 20 years, the advance of the semiconductor technology resulted in the increase of the available number of transistors, which was used to increase the SIMD width and the number of cores
- This direction seems to have reached the dead end with Xeon Phi

So where should we go?

What (I think) we should do

First, we need to understand what we are doing now to develop software for modern microprocessors, and investigate better ways.
In order to achieve high efficiency on modern many-core, wide-SIMD processors, we need to

1. **make efficient use of SIMD units**
2. **use many cores efficiently**
3. **take advantage of cache hierarchy to reduce main memory access**

Problem with SIMD unit

- Unlike the vector processors in 1980s, stride or indirect access have huge performance penalty. Even the penalty of unaligned sequential access cannot be ignored
- Even with very wide SIMD units, x86 microprocessors have difficulties in competing with GPGPUs and other many-core architectures with simpler hardware, in particular in the field of performance per watt.

Problem with many-core architecture

- The lack of hardware support for synchronization, direct core-to-core communication and reduction/broad operation makes fine-grain parallelism difficult (Sunway processor have support for most of these)
- Coherent shared cache consumes large area and electricity.

Problem with cache hierarchy

- **L1 is very small and thus difficult to reuse data**
- **The bandwidth of L2 and L3 (and L4) is generally too low to be useful (better than nothing, but...)**

My impression

Looks very similar to the situation when first-generation RISC processors were developped

- **Architecture has "evolved" for some years**
- **Too many things are done in hardware in ways too complicated**
- **We should let software control hardware**
- **MIPS approach might be necessary, in a more extended form.**

One (old) example — GRAPE-DR

- 512-way SIMD processor, but with local memory for each core
- Within the local memory, stride or indirect access can be done with no penalty
- Local memories are dominant on-chip memory (no large Lx caches). Thus, we have fast access to fairly large memory (much larger than on-chip LLC of modern microprocessors)
- Hierarchical on-chip network with the hardware for broadcast/reduction makes fine-grained parallelism highly efficient (without this it would be difficult to make efficient use of 512 cores)
- Horizontal microcode (almost) exposed to software

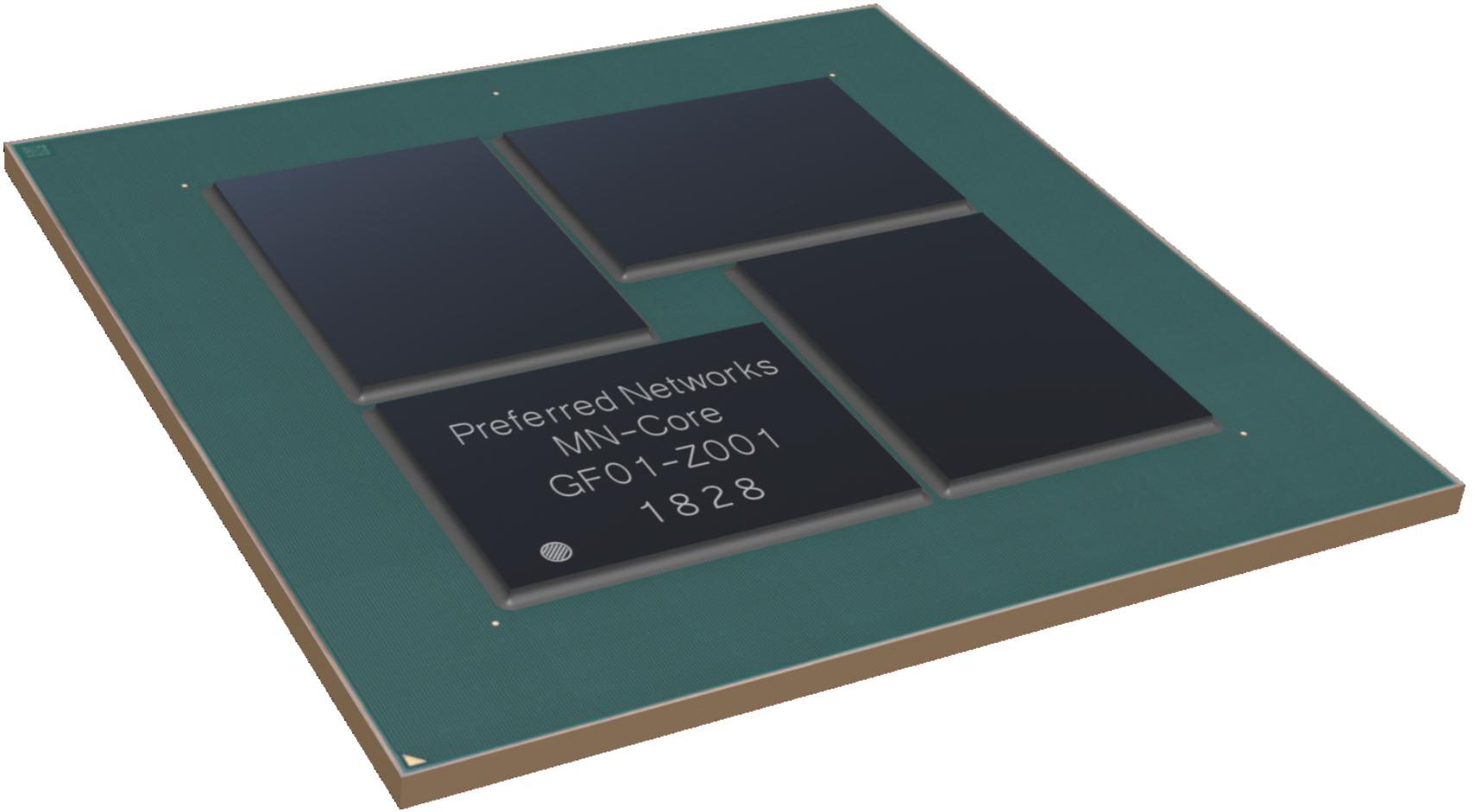
MN-Core (aka GRAPE-PFN2)

- The processor chip PFN (Preferred Networks, a Japanese AI venture) has been developing in collaboration with some of our group in RIKEN R-CCS/Kobe University.
- Goal: Highest performance and highest performance-per-watt for training DNNs (CNNs).
- Planned peak FP16(-equivalent) performance of single card: 524 Tops
- Target power consumption: <500W, > 1Tops/W

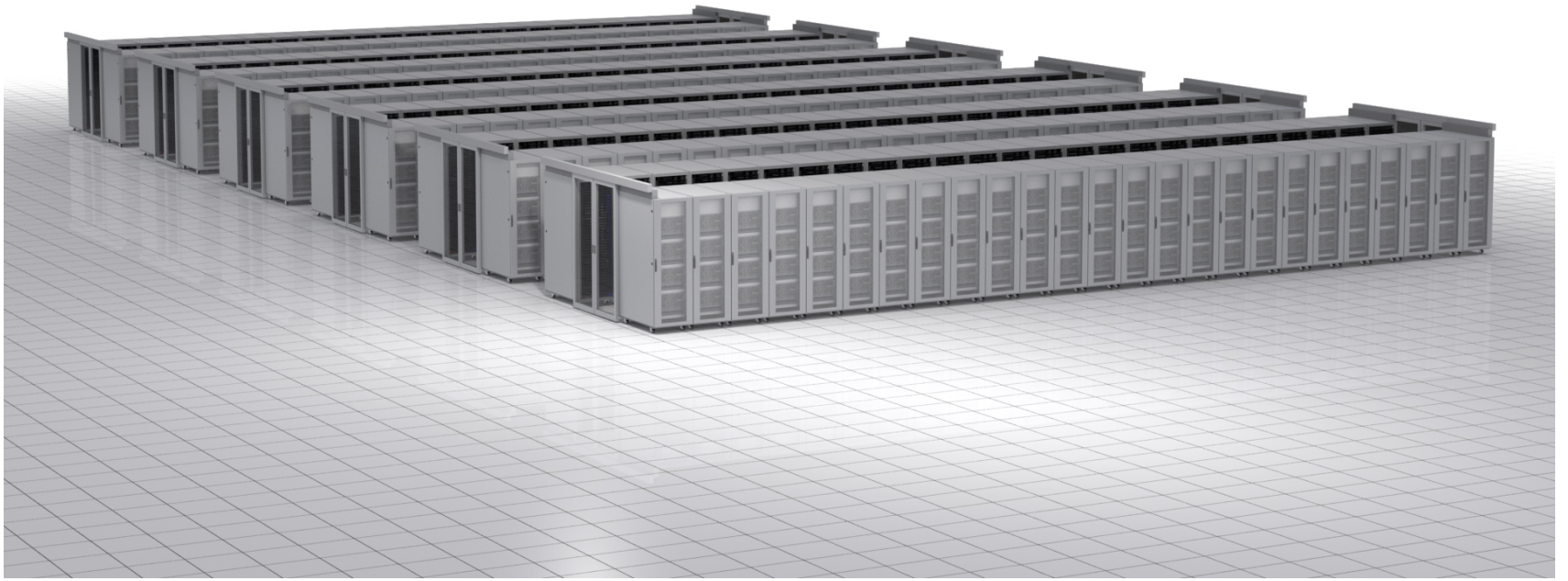
Past history, current status, and future plan.

- **Feb 2016: JM visited PFN at Hongo-3choume**
- **June 2016: Joint application to NEDO ("small" grant, 40MJYE/ye ×2) (PFN moved to Ote-machi)**
- **July 2016: PFN chip project started. Plan for two chips: GPFN1 by NEDO money (40nm, small chip), GPFN2 (12FFC, full-blown) by PFN internal money.**
- **2019 Evaluation of ES chips will be ...**
- **2020 "2EF" system (MN-3) will be ready at JAMSTEC ES site.**

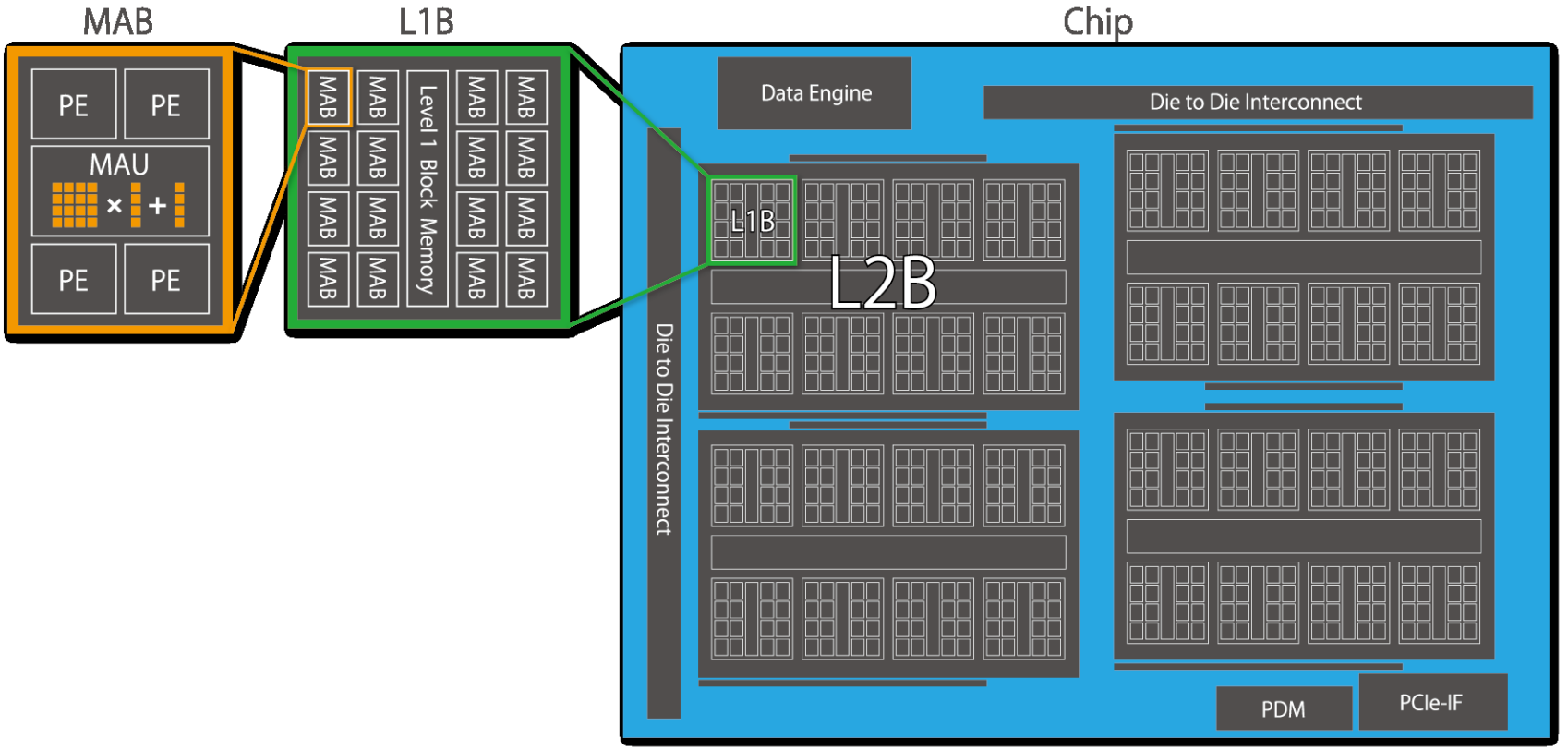
MN-Core



MN-3



GRAPE-PFN2 architecture



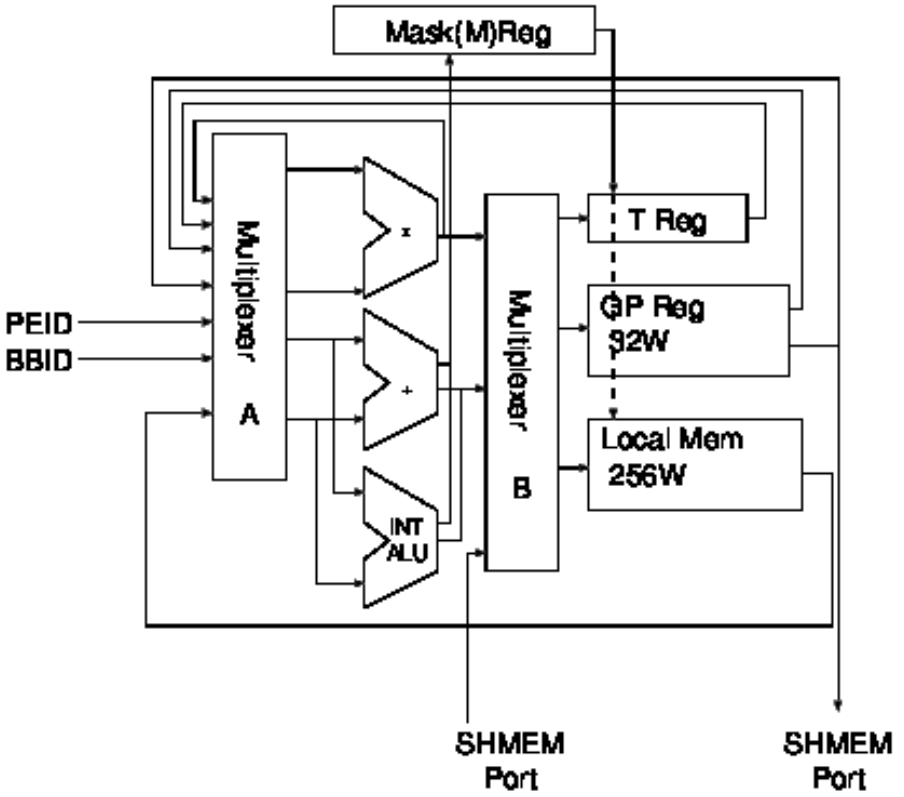
Overview of GPFN2

- One card: One "module", One module: four chips in one package
- One chip: one PCIe interface, DRAM interfaces, four "Level-2 broadcast blocks" (L2Bs)
- One L2B: eight L1Bs
- One L1B: 16 MABs (Matrix Arithmetic Blocks)
- One MAB: four Processor Elements combined to perform DP, SP, or HP matrix-vector multiplication.
- One PE can be also used as scalar processor. We added many special instructions for DL.
- All PE/MAB/L1B/L2B operate on single clock and single instruction stream (card-level SIMD)

GRAPE-DR

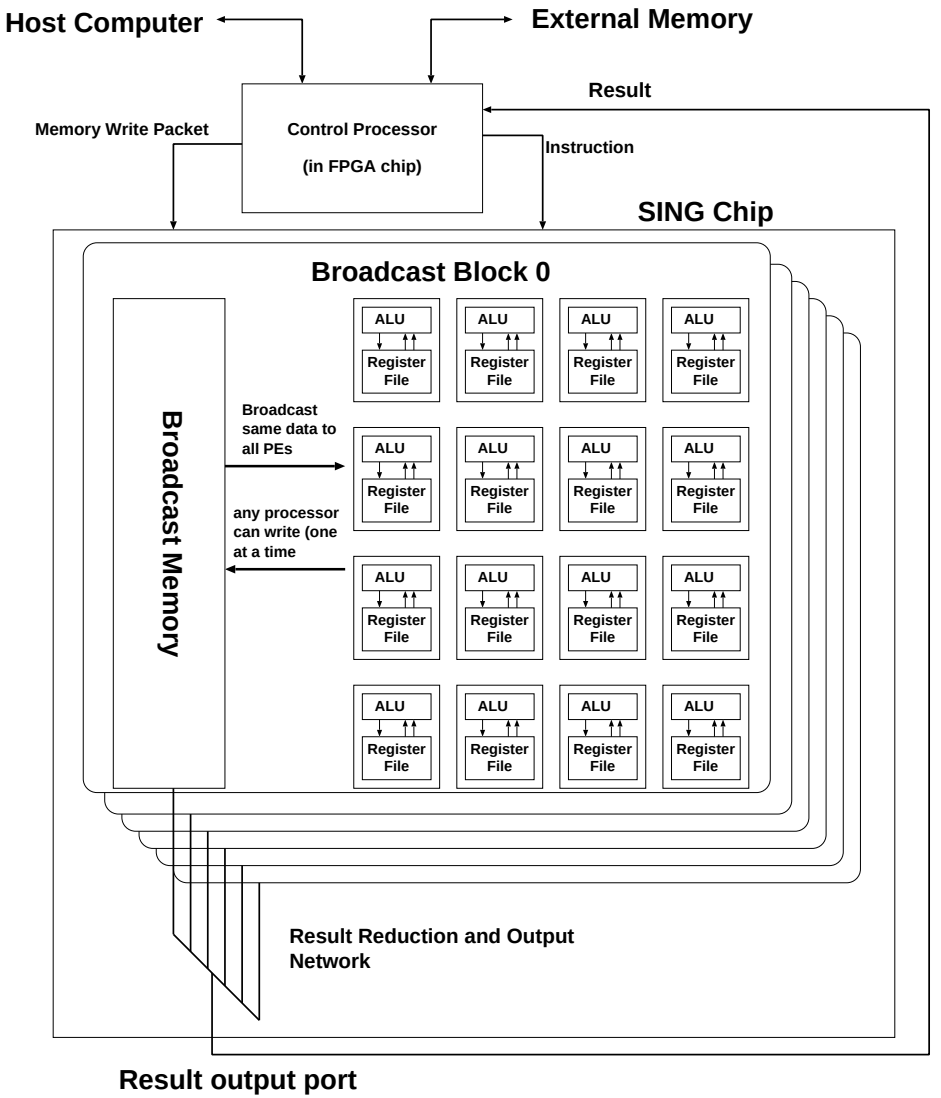
- **Many of the technical details of GRAPE-PFN2 is still undisclosed.**
- **GRAPE-PFN2 is very much a natural extension of GRAPE-DR.**
- **GRAPE-DR project started in 2004 and the machine completed in 2009.**
- **GRAPE-DR chip: completed in 2006, TSMC 90nm, 500MHz, 256 DP Gflops, $\sim$ 4 GF/W.**

GRAPE-DR Processor architecture



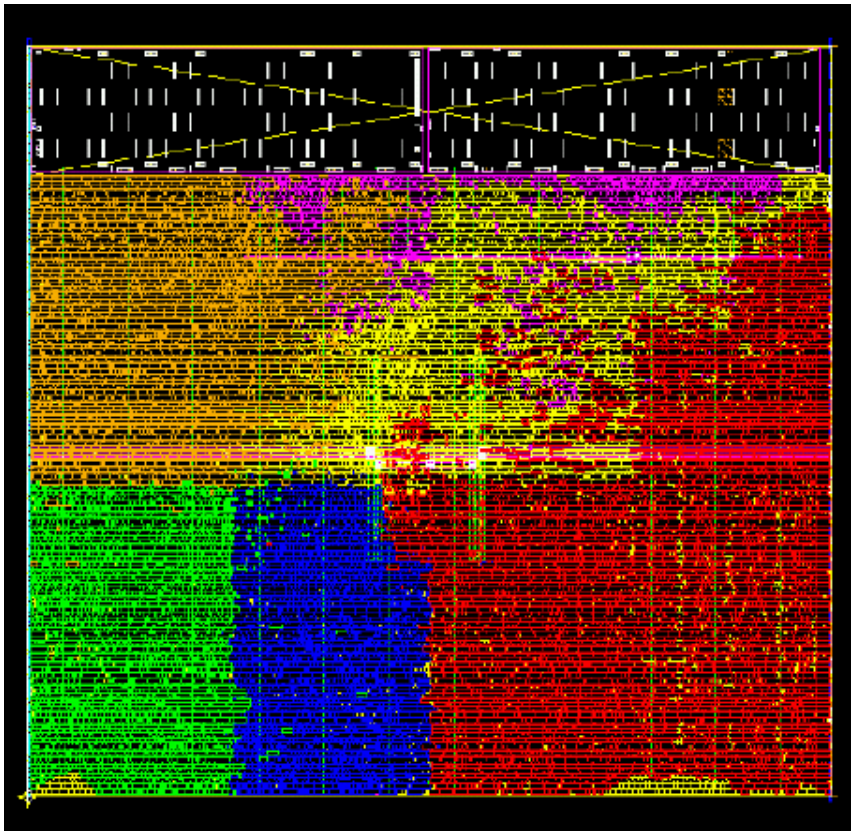
- Float Mult
- Float add/sub
- Integer ALU
- 32-word registers
- 256-word memory
- communication port

Chip architecture



- 32 PEs organized to "broadcast block" (BB)
- BB has shared memory.
- Input data is broadcasted to all BBs.
- Outputs from BBs go through reduction network (sum etc)

PE Layout

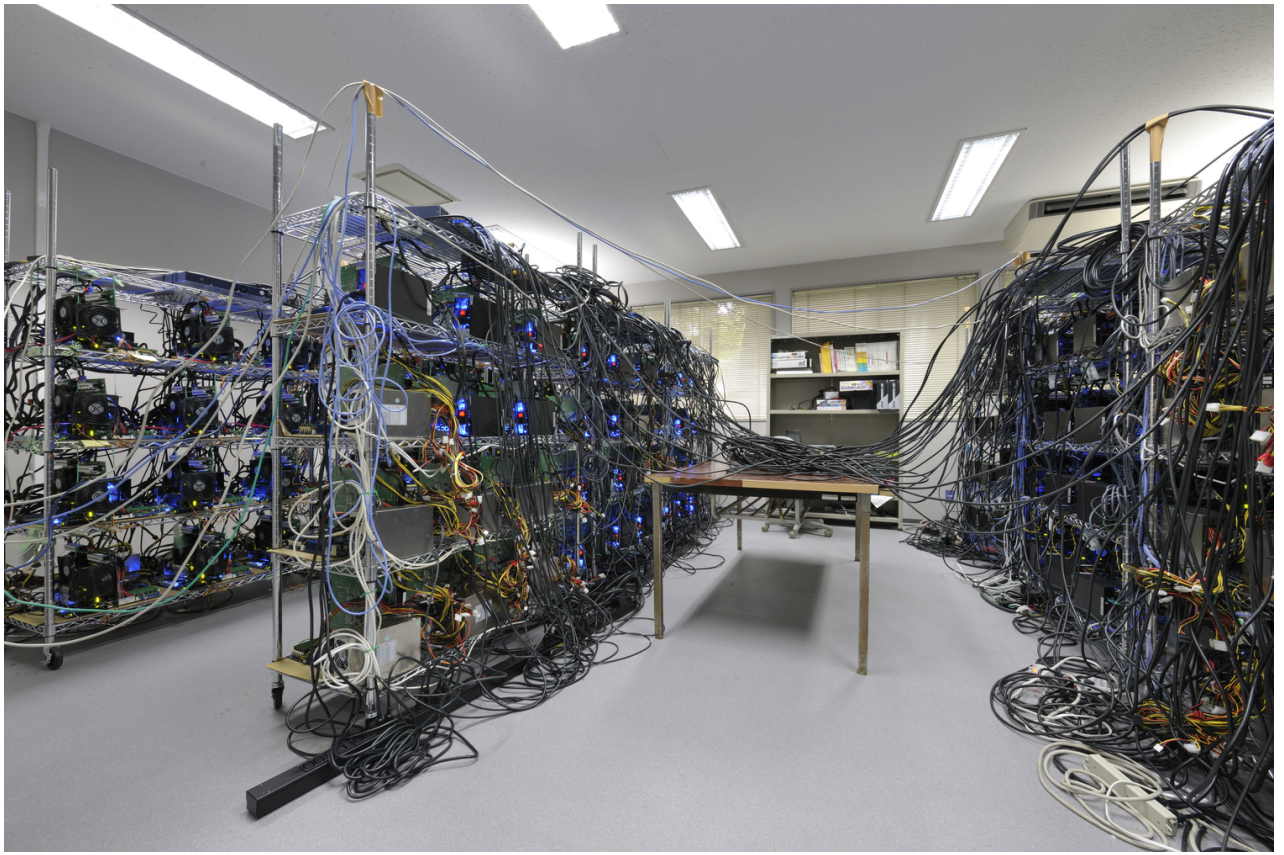


Black: Local Memory
Red: Reg. File
Orange: FMUL
Green: FADD
Blue: IALU

0.7mm by 0.7mm
800K transistors

0.1W@400MHz
800Mflops/400Mflops
peak (SP/DP)

GRAPE-DR cluster system



(As far as I know) Only processor designed in academia listed in Top500 in the last 20 years.

Little Green 500, June 2010

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	815.43	National Astronomical Observatory of Japan	GRAPE-DR accelerator Cluster, Infiniband	28.67
2	773.38	Forschungszentrum Juelich (FZJ)	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
2	773.38	Universitaet Regensburg	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
2	773.38	Universitaet Wuppertal	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
5	536.24	Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw	BladeCenter QS22 Cluster, PowerXCell 8i 4.0 Ghz, Infiniband	34.63

#1: GRAPE-DR,
#2: QPACE: Ge
QCD machine
#9: NVIDIA Ferm

Changes made from GRAPE-DR

- Second layer of on-chip tree network
- Integration of PCIe and DRAM interface
- Addition of MAB
- Much larger memory
- Many other changes in on-cip network
- Design optimized to DNN/CNN
  (both inference and learning)

Some numbers

- **Number of MABs and PEs: 512 (2048) per chip, 2048 (8192) per module.**
- **32.8TF (DP), 132TF (SP) and 524Tops (HP)**
- **Memory bandwidth: (not yet open)**
- **Link to the host PC: PCIe**
- **On-board DRAM: 32GB**

Writing software for GRAPE-DR/MN-Core

- You need to take care of moving data between DRAM, on-chip network hierarchy, and processor elements.
- However, that is what you need to think about to achieve high performance on cache-based processor anyway.
- To have explicit control on hardware makes the tuning of the performance rather easy, since the performance is predictable.
- GRAPE-DR has no hardware for interlock. Number of instructions= number of cycles.
- Local memory has B/F=8. So it is "straightforward" to write efficient kernels, once data are in local memory.

Summary

- We have been working on FDPS, a high-performance framework for developing particle-based simulation codes
- Current version (5.0) supports user programs in Fortran, C, or other languages, and can make use of GPU and other accelerators efficiently.
- We are also working on processors for HPC (mainly DL...) with design concept quite different from that of modern microprocessors.
- So far, the development goes well.