MAX PLANCK INSTITUTE
FOR DYNAMICS OF COMPLEX
TECHNICAL SYSTEMS
MAGDEBURG

COMPUTATIONAL METHODS IN
SYSTEMS AND CONTROL THEORY

GPU Accelerated Gauss-Jordan Elimination on the OpenPOWER platform – A case study

Martin Köhler

March 8, 2017

GAMM Annual Meeting
Scientific Computing Section

Motivation

Matrix-Sign-Function:

We consider the Newton iteration to compute Matrix-Sign-Function $X_\infty := \operatorname{sign}(A)$ of a matrix $A \in \mathbb{R}^{n\times n}$:

$$X_{k+1} = \frac{1}{2}\left(\mu_k X_k + \mu_k^{-1} X_k^{-1}\right), \quad X_0 = A,$$

where μ_k is a scaling factor, typically $\mu_k := |\det X_k|^{-\frac{1}{n}}$.

Motivation

Matrix-Sign-Function:

We consider the Newton iteration to compute Matrix-Sign-Function $X_\infty := \operatorname{sign}(A)$ of a matrix $A \in \mathbb{R}^{n\times n}$:

$$X_{k+1} = \frac{1}{2}\left(\mu_k X_k + \mu_k^{-1} X_k^{-1}\right), \quad X_0 = A,$$

where μ_k is a scaling factor, typically $\mu_k := |\det X_k|^{-\frac{1}{n}}$.

Applications:

- Computation of invariant subspaces of A
- Solution of the standard Lyapunov equation
- Solution of the standard Riccati equation

Motivation

Matrix-Sign-Function:

We consider the Newton iteration to compute Matrix-Sign-Function $X_\infty := \operatorname{sign}(A, B)$ of a matrix pencil $(A, B) \in \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times n}$:

$$X_{k+1} = \frac{1}{2}\left(\mu_k X_k + \mu_k^{-1} B X_k^{-1} B\right), \quad X_0 = A,$$

where μ_k is a scaling factor, typically $\mu_k := \left(\frac{|\det X_k|}{|\det B|}\right)^{-\frac{1}{n}}$.

Applications:

- Computation of deflating subspaces of (A, B)
- Solution of the generalized Lyapunov equation
- Solution of the generalized Riccati equation

Motivation

Matrix-Sign-Function:

We consider the Newton iteration to compute Matrix-Sign-Function $X_\infty := \operatorname{sign}(A, B)$ of a matrix pencil $(A, B) \in \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times n}$:

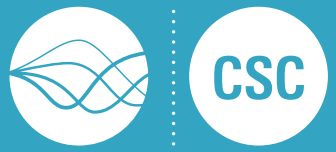
$$X_{k+1} = \frac{1}{2}\left(\mu_k X_k + \mu_k^{-1} B X_k^{-1} B\right), \quad X_0 = A,$$

where μ_k is a scaling factor, typically $\mu_k := \left(\frac{|\det X_k|}{|\det B|}\right)^{-\frac{1}{n}}$.

Applications:

- Computation of deflating subspaces of (A, B)
- Solution of the generalized Lyapunov equation
- Solution of the generalized Riccati equation

Need to compute A^{-1} or to solve $AY = B$ with many right hand sides.



Motivation

Computation of A^{-1}: (as in LAPACK/MAGMA)

1. Compute LU decomposition of A **cost:** $\frac{2}{3}n^3$
2. Inversion of U **cost:** $\frac{1}{3}n^3$
3. Solution of $LA^{-1} = U^{-1}$ **cost:** n^3

Motivation

Computation of A^{-1}: (as in LAPACK/MAGMA)

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Inversion of U — **cost:** $\frac{1}{3}n^3$
3. Solution of $LA^{-1} = U^{-1}$ — **cost:** n^3

Solution of $AY = B$:

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Solution of $LZ = B$ — **cost:** n^3
3. Solution of $UY = Z$ — **cost:** n^3

Motivation

Computation of A^{-1}: (as in LAPACK/MAGMA)

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Inversion of U — **cost:** $\frac{1}{3}n^3$
3. Solution of $LA^{-1} = U^{-1}$ — **cost:** n^3

Solution of $AY = B$:

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Solution of $LZ = B$ — **cost:** n^3
3. Solution of $UY = Z$ — **cost:** n^3

- All three steps are of cubic complexity.
- Solving with triangular matrices is not well suited for (massively) parallel architectures.

Motivation

Computation of A^{-1}: (as in LAPACK/MAGMA)

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Inversion of U — **cost:** $\frac{1}{3}n^3$
3. Solution of $LA^{-1} = U^{-1}$ — **cost:** n^3

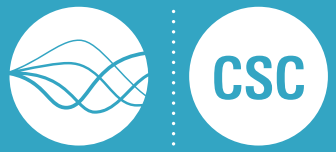
Solution of $AY = B$:

1. Compute LU decomposition of A — **cost:** $\frac{2}{3}n^3$
2. Solution of $LZ = B$ — **cost:** n^3
3. Solution of $UY = Z$ — **cost:** n^3

In both cases, we want combine all three steps into a single, easily parallelizable, algorithm.

$\rightarrow$ **Gauss-Jordan Elimination**

- All three steps are of cubic complexity.
- Solving with triangular matrices is not well suited for (massively) parallel architectures.



Motivation – Hardware

OpenPOWER

Industry alliance (Google, IBM, Canonical, RedHat, Mellanox, Nvidia,...) to develop a customizable server platform for data centers, high performance computing, . . . on top of Open Source philosophy.

Motivation – Hardware

OpenPOWER

Industry alliance (Google, IBM, Canonical, RedHat, Mellanox, Nvidia,...) to develop a customizable server platform for data centers, high performance computing, . . . on top of Open Source philosophy.

IBM Power System 822LC for HPC

- 2x IBM POWER8 CPUs
  (each: 10 Cores, 8-way SMT, 10x 512Kb L2 Cache, 10x 8MB L3 Cache, 4.00GHz)
- 256GB DDR4 memory, bandwidth: 230 GB/s
- 2x Nvidia Tesla P100 SXM2 accelerators with 16GB HBM2 memory
- NVLink CPU-GPU interconnect, bidirectional bandwidth: 80 GB/s
- Theoretical peak performance [TFlops/s]: 10.6 (DP), 21.2 (SP), 42.4 (HP)
- Staging system for the upcoming POWER 9 + Nvidia Volta architecture

Motivation – Hardware

OpenPOWER

Industry alliance (Google, IBM, Canonical, RedHat, Mellanox, Nvidia,...) to develop a customizable server platform for data centers, high performance computing, . . . on top of Open Source philosophy.

IBM Powe

- 2x IB
  (each
  4.00G
- 256G
- 2x Nv
- NVLink CPU-GPU interconnect, bidirectional bandwidth: 80 GB/s
- Theoretical peak performance [TFlops/s]: 10.6 (DP), 21.2 (SP), 42.4 (HP)
- Staging system for the upcoming POWER 9 + Nvidia Volta architecture

- **CPU performance comparable with a 2x 8 Core Intel Haswell v3**
- **GPU performance $\approx$ 5 times higher as of Kepler generation GPUs (K20 or a single GPU on K80)**

Motivation – Hardware

OpenPOWER

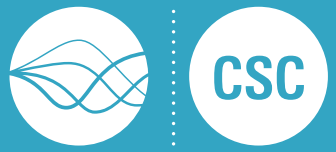
Industry alliance (Google, IBM, Canonical, RedHat, Mellanox, Nvidia,...) to develop a customizable server platform for data centers, high performance computing, . . . on top of Open Source philosophy.

IBM Powe

- 2x IB
  (each
  4.00G
- 256G
- 2x Nv
- NVLink CPU-GPU interconnect, bidirectional bandwidth: 80 GB/s
- Theoretical peak performance [TFlops/s]: 10.6 (DP), 21.2 (SP), 42.4 (HP)
- Staging system for the upcoming POWER 9 + Nvidia Volta architecture

- **CPU performance comparable with a 2x 8 Core Intel Haswell v3**
- **GPU performance $\approx$ 5 times higher as of Kepler generation GPUs (K20 or a single GPU on K80)**

Increase the GPU load on the OpenPOWER platform to obtain high performance.



Gauss-Jordan Elimination

The Gauss-Jordan Elimination algorithm is mostly known to invert matrices with $2n^3$ flops.

Solving a linear system with n right hand sides costs:

Gauss-Jordan Elimination

The Gauss-Jordan Elimination algorithm is mostly known to invert matrices with $2n^3$ flops.

Solving a linear system with n right hand sides costs:

$\rightarrow$ $2n^3 + 2n^3$ flops using classical Gauss-Jordan Elimination.

Gauss-Jordan Elimination

The Gauss-Jordan Elimination algorithm is mostly known to invert matrices with $2n^3$ flops.

Solving a linear system with n right hand sides costs:

$\rightarrow$ $2n^3 + 2n^3$ flops using classical Gauss-Jordan Elimination.

$\rightarrow$ $\frac{2}{3}n^3 + 2n^3$ flops using LU decomposition.

Gauss-Jordan Elimination

The Gauss-Jordan Elimination algorithm is mostly known to invert matrices with $2n^3$ flops.

Solving a linear system with n right hand sides costs:

$\rightarrow$ $2n^3 + 2n^3$ flops using classical Gauss-Jordan Elimination.
$\rightarrow$ $\frac{2}{3}n^3 + 2n^3$ flops using LU decomposition.

Modify the Gauss-Jordan Elimination scheme such that:

- the multiplication with the inverse is done implicitly,
- and the inverse is not accumulated.

Gauss-Jordan Elimination

The Gauss-Jordan Elimination algorithm is mostly known to invert matrices with $2n^3$ flops.

Solving a linear system with n right hand sides costs:

$\rightarrow$ $2n^3 + 2n^3$ flops using classical Gauss-Jordan Elimination.
$\rightarrow$ $\frac{2}{3}n^3 + 2n^3$ flops using LU decomposition.

Modify the Gauss-Jordan Elimination scheme such that:

- the multiplication with the inverse is done implicitly,
- and the inverse is not accumulated.

Goals:

- Reduce the number of necessary flops
- Increase memory access locality

Gauss-Jordan Elimination

We consider the *augmented matrix D*

$$D := \left[\begin{array}{c|c|c} A & | & B \end{array}\right] = \left[\begin{array}{ccc|ccc} a_{11} & \cdots & a_{1m} & b_{11} & \cdots & b_{1n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mm} & b_{m1} & \cdots & b_{mn} \end{array}\right]$$

and apply a set of transformations $\tilde{G}_i$ from the left such that we obtain:

$$\underbrace{\tilde{G}_n \cdots \tilde{G}_2 \tilde{G}_1}_{A^{-1}} D = \left[\begin{array}{ccc|ccc} 1 & & & y_{11} & \cdots & y_{1n} \\ & \ddots & & \vdots & \vdots & \vdots \\ & & 1 & y_{m1} & \cdots & y_{mn} \end{array}\right].$$

Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss transformation G_i:

$$G_i = \begin{bmatrix} 1 & & & -\frac{d_{1i}}{d_{ii}} & & \\ & \ddots & & \vdots & & \\ & & 1 & -\frac{d_{(i-1)i}}{d_{ii}} & & \\ & & & \frac{1}{d_{ii}} & & \\ & & & -\frac{d_{(i+1)i}}{d_{ii}} & 1 & \\ & & & \vdots & & \ddots \\ & & & -\frac{d_{mi}}{d_{ii}} & & 1 \end{bmatrix},$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \ldots \tilde{G}_1$.

Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss tran

Example – Applying $\tilde{G}_i$:

$$D = \left[\begin{array}{cccc|ccc} a_{11} & a_{12} & a_{13} & a_{14} & b_{11} & \dots & b_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & b_{21} & \dots & b_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & b_{31} & \dots & b_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & b_{41} & \dots & b_{4n} \end{array}\right]$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \dots \tilde{G}_1$.

Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss tran

Example – Applying $\tilde{G}_i$:

$$\tilde{G}_1 D = \left[\begin{array}{cccc|ccc} 1 & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} & b_{11}^{(1)} & \dots & b_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} & b_{21}^{(1)} & \dots & b_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} & b_{31}^{(1)} & \dots & b_{3n}^{(1)} \\ 0 & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} & b_{41}^{(1)} & \dots & b_{4n}^{(1)} \end{array}\right]$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \dots \tilde{G}_1$.

Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss tran

Example – Applying $\tilde{G}_i$:

$$\tilde{G}_2 \tilde{G}_1 D = \left[\begin{array}{cccc|ccc} 1 & 0 & a_{13}^{(2)} & a_{14}^{(2)} & b_{11}^{(2)} & \dots & b_{1n}^{(2)} \\ 0 & 1 & a_{23}^{(2)} & a_{24}^{(2)} & b_{21}^{(2)} & \dots & b_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(2)} & a_{34}^{(2)} & b_{31}^{(2)} & \dots & b_{3n}^{(2)} \\ 0 & 0 & a_{43}^{(2)} & a_{44}^{(2)} & b_{41}^{(2)} & \dots & b_{4n}^{(2)} \end{array}\right]$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \dots \tilde{G}_1$.

Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss tran

Example – Applying $\tilde{G}_i$:

$$\tilde{G}_3 \tilde{G}_2 \tilde{G}_1 D = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & a_{14}^{(3)} & b_{11}^{(3)} & \dots & b_{1n}^{(3)} \\ 0 & 1 & 0 & a_{24}^{(3)} & b_{21}^{(3)} & \dots & b_{2n}^{(3)} \\ 0 & 0 & 1 & a_{34}^{(3)} & b_{31}^{(3)} & \dots & b_{3n}^{(3)} \\ 0 & 0 & 0 & a_{44}^{(3)} & b_{41}^{(3)} & \dots & b_{4n}^{(3)} \end{array}\right]$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \dots \tilde{G}_1$.

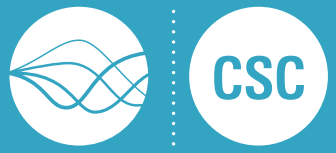
Gauss-Jordan Elimination

We define $\tilde{G}_i = G_i P_i$ as product of a row permutation P_i and a Gauss tran

Example – Applying $\tilde{G}_i$:

$$\underbrace{\tilde{G}_4 \tilde{G}_3 \tilde{G}_2 \tilde{G}_1}_{A^{-1}} D = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & y_{11} & \dots & y_{1n} \\ 0 & 1 & 0 & 0 & y_{21} & \dots & y_{2n} \\ 0 & 0 & 1 & 0 & y_{31} & \dots & y_{3n} \\ 0 & 0 & 0 & 1 & y_{41} & \dots & y_{4n} \end{array}\right]$$

where d_{kl} are the entries in D after applying $\tilde{G}_{i-1} \dots \tilde{G}_1$.



Blocked Algorithm

The application of G_i can be replaced by a rank-1 update and a row scaling in order to work in-place:

$$D := D - \frac{1}{d_{ii}} \left(d_{1i}, \cdots, d_{(i-1)i}, 0, d_{(i+1)i}, \ldots, d_{mi}\right)^T D_{i,.}$$

$$D_{i,.} := \frac{1}{d_{ii}} D_{i,.}.$$

Blocked Algorithm

The application of G_i can be replaced by a rank-1 update and a row scaling in order to work in-place:

$$D := D - \frac{1}{d_{ii}} \left(d_{1i}, \cdots, d_{(i-1)i}, 0, d_{(i+1)i}, \ldots, d_{mi} \right)^T D_{i,\cdot}$$

$$D_{i,\cdot} := \frac{1}{d_{ii}} D_{i,\cdot}.$$

By repartitioning D into blocks:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & b_1 \\ \hline A_{21} & A_{22} & A_{23} & b_2 \\ \hline A_{31} & A_{32} & A_{33} & b_3 \end{array}\right],$$

where $A_{22} \in \mathbb{R}^{N_B \times N_B}$, we transform the rank-1 update into a rank-N_B update.

Blocked Algorithm

By repartitioning D into blocks:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & b_1 \\ \hline A_{21} & A_{22} & A_{23} & b_2 \\ \hline A_{31} & A_{32} & A_{33} & b_3 \end{array}\right],$$

where $A_{22} \in \mathbb{R}^{N_B \times N_B}$, we transform the rank-1 update into a rank-N_B update:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & 0 & A_{13} & b_1 \\ \hline 0 & 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} & b_3 \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \begin{bmatrix} A_{21} & I_{N_B} & A_{23} & b_2 \end{bmatrix}.$$

Blocked Algorithm

By repartitioning D into blocks:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & b_1 \\ \hline A_{21} & A_{22} & A_{23} & b_2 \\ \hline A_{31} & A_{32} & A_{33} & b_3 \end{array}\right],$$

where $A_{22} \in \mathbb{R}^{N_B \times N_B}$, we transform the rank-1 update into a rank-N_B update:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & 0 & A_{13} & b_1 \\ \hline 0 & 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} & b_3 \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \begin{bmatrix} A_{21} & I_{N_B} & A_{23} & b_2 \end{bmatrix}.$$

Algorithm mainly relies on rank-N_B updates.

Blocked Algorithm

By repartitioning D into blocks:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & b_1 \\ \hline A_{21} & A_{22} & A_{23} & b_2 \\ \hline A_{31} & A_{32} & A_{33} & b_3 \end{array}\right],$$

where $A_{22} \in \mathbb{R}^{N_B \times N_B}$, we transform the rank-1 update into a rank-N_B update:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & 0 & A_{13} & b_1 \\ \hline 0 & 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} & b_3 \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \begin{bmatrix} A_{21} & I_{N_B} & A_{23} & b_2 \end{bmatrix}.$$

Algorithm mainly relies on rank-N_B updates.

Pivoting is integrated by LU decomposition of $\begin{bmatrix} A_{22}^T & A_{32}^T \end{bmatrix}^T$.

Blocked Algorithm

By re

wher :

- If only the solution $AY = B$ required, the update reduces to:

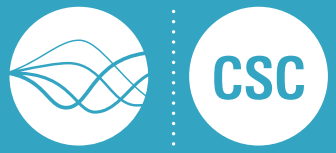
$$\left[\begin{array}{c|c} A_{13} & b_1 \\ \hline 0 & 0 \\ \hline A_{33} & b_3 \end{array}\right] \leftarrow \left[\begin{array}{c|c} A_{13} & b_1 \\ \hline 0 & 0 \\ \hline A_{33} & b_3 \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \left[A_{23} \;\; b_2\right].$$

  → **Reduces the flop count to $n^3 + 2n^3$.**

- If only the inverse A^{-1} is necessary we obtain:

$$\left[\begin{array}{c|c|c} A_{11} & 0 & A_{13} \\ \hline 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} \end{array}\right] \leftarrow \left[\begin{array}{c|c|c} A_{11} & 0 & A_{13} \\ \hline 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \left[\begin{array}{ccc} A_{21} & I_{N_B} & A_{23} \end{array}\right].$$

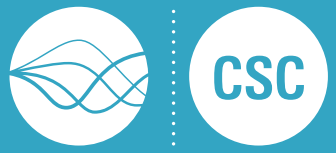
  → **Same flop count $2n^3$ as with LU decomposition.**



Multi-GPU Implementation

The algorithm has the following properties:

- the computation of the panel matrix H works inside a block column,
- the rank-N_B update with the matrix H is a `GEMM` operation.



Multi-GPU Implementation

The algorithm has the following properties:

- the computation of the panel matrix H works inside a block column,
- the rank-N_B update with the matrix H is a `GEMM` operation.

$\rightarrow$ Data layout must cover the computation of the panel matrix H and the `GEMM` operation.

Multi-GPU Implementation

The algorithm has the following properties:

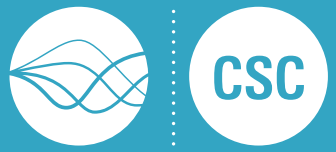
- the computation of the panel matrix H works inside a block column,
- the rank-N_B update with the matrix H is a `GEMM` operation.

$\rightarrow$ Data layout must cover the computation of the panel matrix H and the `GEMM` operation.

Data Layout

Only $\mathcal{O}(1)$ GPUs in one server available $\rightarrow$ **Column Block Cyclic (CBC)** distribution of the matrix D:

$$D := \left[\begin{array}{|c|c|c|c|} \hline \text{GPU 1} & \text{GPU 2} & \text{GPU 1} & \text{GPU 2} \\ \hline \end{array}\right]$$



Multi-GPU Implementation

Basic GPU Workflow

After separation into CPU-aware and GPU-aware operations we have to:

1. Compute the panel matrix H on the host CPU,
2. Copy the panel matrix H to all GPUs,
3. Perfom the rank-N_B update in parallel on the distributed matrix D.

Multi-GPU Implementation

Basic GPU Workflow

After separation into CPU-aware and GPU-aware operations we have to:

1. Compute the panel matrix H on the host CPU,
2. Copy the panel matrix H to all GPUs,
3. Perfom the rank-N_B update in parallel on the distributed matrix D.

Parallel Rank-N_B Update

$$\left[\begin{array}{c|c} A_{13} & b_1 \\ \hline 0 & 0 \\ \hline A_{33} & b_3 \end{array}\right] \leftarrow \left[\begin{array}{c|c} A_{13} & b_1 \\ \hline 0 & 0 \\ \hline A_{33} & b_3 \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \left[\begin{array}{cc} A_{23} & b_2 \end{array}\right].$$

Multi-GPU Implementation

Basic GPU Workflow

After separation into CPU-aware and GPU-aware operations we have to:

1. Compute the panel matrix H on the host CPU,
2. Copy the panel matrix H to all GPUs,
3. Perfom the rank-N_B update in parallel on the distributed matrix D.

Parallel Rank-N_B Update

Multi-GPU Implementation

Look-Ahead and Asynchronous Operation

We split right part of D into

$$\left[\begin{array}{c|c} A_{13} & b_1 \\ \hline A_{23} & b_2 \\ \hline A_{33} & b_3 \end{array}\right] := \left[\begin{array}{c|c|c} \hat{A}_{13} & \bar{A}_{13} & b_1 \\ \hline \hat{A}_{23} & \bar{A}_{23} & b_2 \\ \hline \hat{A}_{33} & \bar{A}_{33} & b_3 \end{array}\right],$$

where $\hat{A}_{23} \in \mathbb{R}^{N_B \times N_B}$ and perform the update in two steps as

$$\left[\begin{array}{c} \hat{A}_{13} \\ \hline 0 \\ \hline \hat{A}_{33} \end{array}\right] \leftarrow \left[\begin{array}{c} \hat{A}_{13} \\ \hline 0 \\ \hline \hat{A}_{33} \end{array}\right] + H\hat{A}_{23} \qquad \text{(Look-Ahead GEMM)}$$

and

$$\left[\begin{array}{c|c} \bar{A}_{13} & b_1 \\ \hline 0 & 0 \\ \hline \bar{A}_{33} & b_3 \end{array}\right] \leftarrow \left[\begin{array}{c|c} \bar{A}_{13} & b_1 \\ \hline 0 & 0 \\ \hline \bar{A}_{33} & b_3 \end{array}\right] + H\left[\bar{A}_{23} \quad b_2\right]. \qquad \text{(Remaining GEMM)}$$

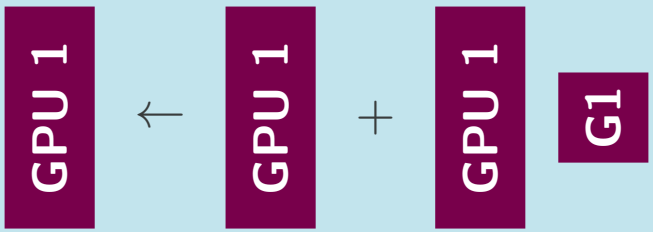
Multi-GPU Implementation

Look-Ahead and Asynchronous Operation

We split right part of D into

$$\left[\begin{array}{c|c} A_{13} & b_1 \\ \hline A_{23} & b_2 \\ \hline A_{33} & b_3 \end{array}\right] := \left[\begin{array}{c|c|c} \hat{A}_{13} & \bar{A}_{13} & b_1 \\ \hline \hat{A}_{23} & \bar{A}_{23} & b_2 \\ \hline \hat{A}_{33} & \bar{A}_{33} & b_3 \end{array}\right],$$

where $\hat{A}_{23} \in \mathbb{R}^{N_B \times N_B}$ and perform the update in two steps as



and

Multi-GPU Implementation

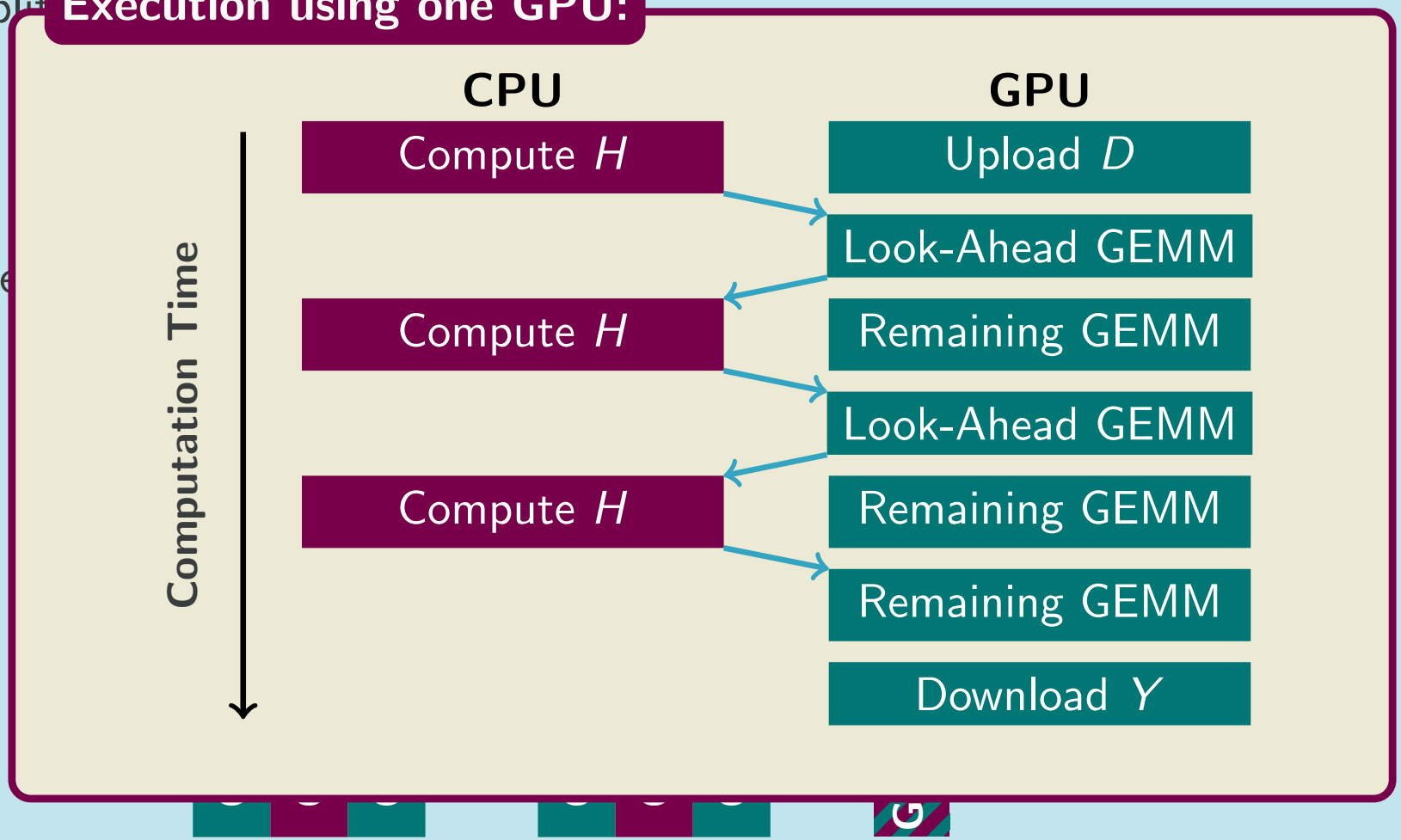
Look-Ahead and Asynchronous Operation

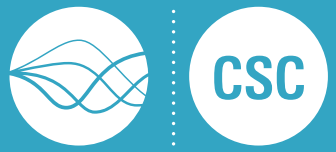
We spli

where

and

Execution using one GPU:





Multi-GPU Implementation

Further Caveats

General

- We have to use **row major** on the GPUs to **reduce** the number of **cache misses** during pivoting.
  $\rightarrow$ The algorithm works implicitly on the transpose of the matrix D.
- All **memory** locations on the host need to be **page-aligned** and **page-locked**.

Multi-GPU Implementation

Further Caveats

General

- We have to use **row major** on the GPUs to **reduce** the number of **cache misses** during pivoting.
  $\rightarrow$ The algorithm works implicitly on the transpose of the matrix D.
- All **memory** locations on the host need to be **page-aligned** and **page-locked**.

Performance Shift on OpenPOWER:

- Performance gap between CPUs and GPUs: $\approx 20\times$
- Higher bandwidths between main memory, CPU, GPU, and GPU memory

$\rightarrow$ Panel preparation on the host is slow, even with multi-threaded BLAS.

Multi-GPU Implementation

Further

Fine grained computation of H:

Input: Current Panel $\left[A_{12}^T\ A_{22}^T\ A_{32}^T\right]^T$

Output: Update matrix H

1: Compute the LU decomposition

$$\begin{bmatrix} L_1 \\ L_2 \end{bmatrix} U = P \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix}$$

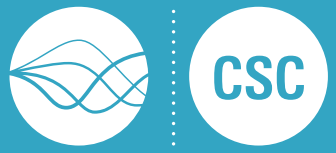
2: Enqueue the permutation P on the devices.

3: Enqueue the preparation of the rank-N_B updates on the devices.

4: Compute

$$H := \begin{bmatrix} -A_{12}U^{-1}L_1^{-1} \\ U_1^{-1}L^{-1} \\ -L_2L_1^{-1} \end{bmatrix}$$

5: Upload H to the devices.



Numerical Results

Hardware and Software Environment

OpenPOWER 8 System

- Hardware as given in the Motivation with 2x P100 accelerators
- CentOS 7.3 for ppc64le with custom 4.8.6 Linux Kernel
- IBM XLC 13.1.5 and IBM XLF 15.1.5 compilers
- CUDA 8.0
- IBM ESSL 5.5 as BLAS and LAPACK library on the host

Numerical Results

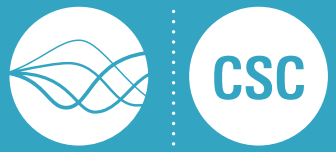
Hardware and Software Environment

OpenPOWER 8 System

- Hardware as given in the Motivation with 2x P100 accelerators
- CentOS 7.3 for ppc64le with custom 4.8.6 Linux Kernel
- IBM XLC 13.1.5 and IBM XLF 15.1.5 compilers
- CUDA 8.0
- IBM ESSL 5.5 as BLAS and LAPACK library on the host

"Old" Reference System

- 2x Intel Xeon E5-2640v3 (8 Cores, 8x 256kB L2 Cache, 20MB L3 Cache)
- 64 GB DDR3 memory
- 2x Nvidia Tesla K20m accelerators
- CentOS 7.3 with Intel Parallel Studio 2017.1 including MKL 2017.1
- CUDA 8.0



Numerical Results

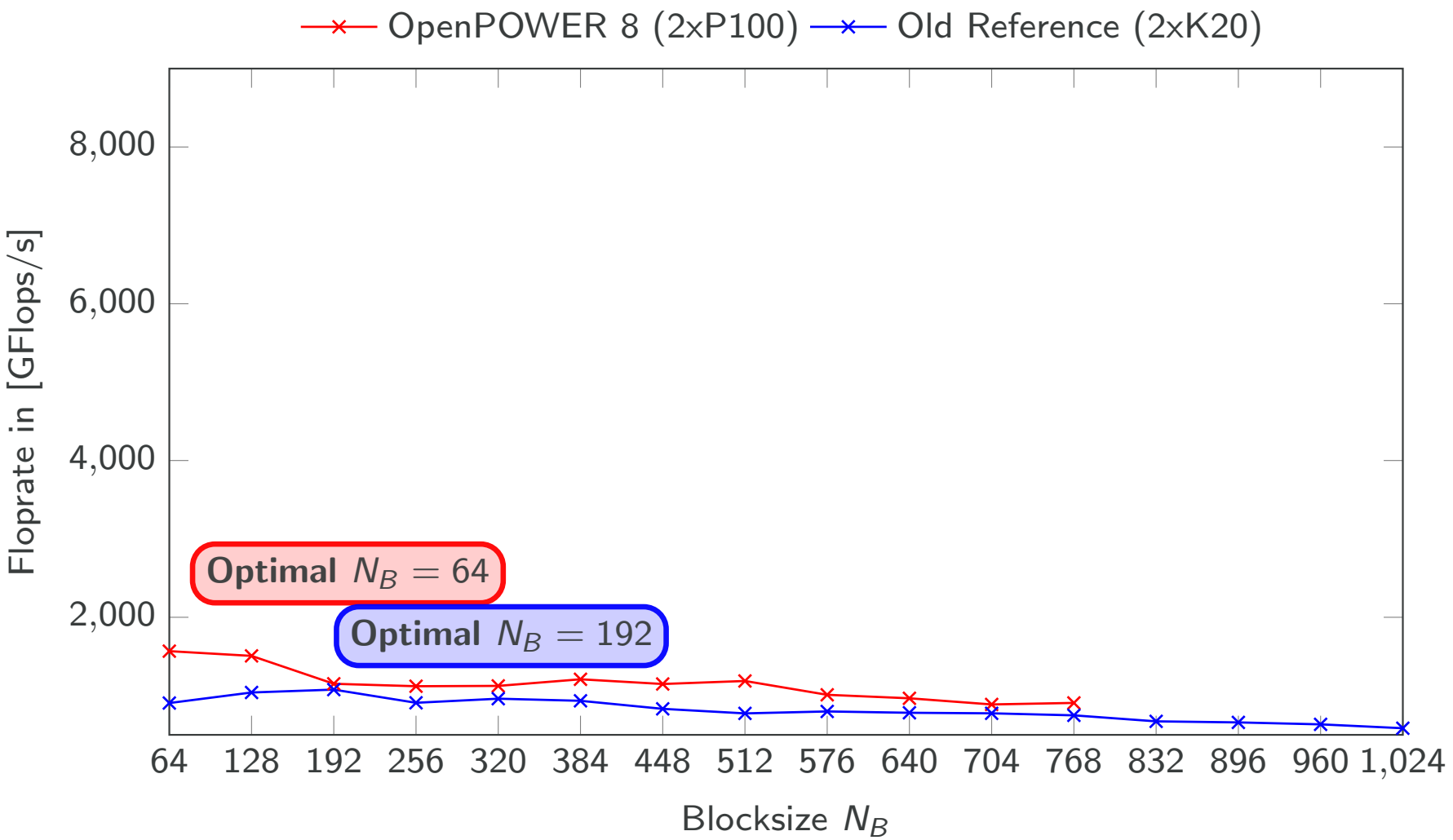
Solution of the linear system:

- Random matrix $A \in \mathbb{R}^{n \times n}$ with $n = 1024, 2048, \ldots$
- Right hand side $B \in \mathbb{R}^{n \times n}$ as $B := A \cdot \texttt{ones}(n, n)$
- Block size varying from 64 to 768(P100) / 1024(K20)
- IEEE Double Precision

Numerical Results

Optimal Blocksize

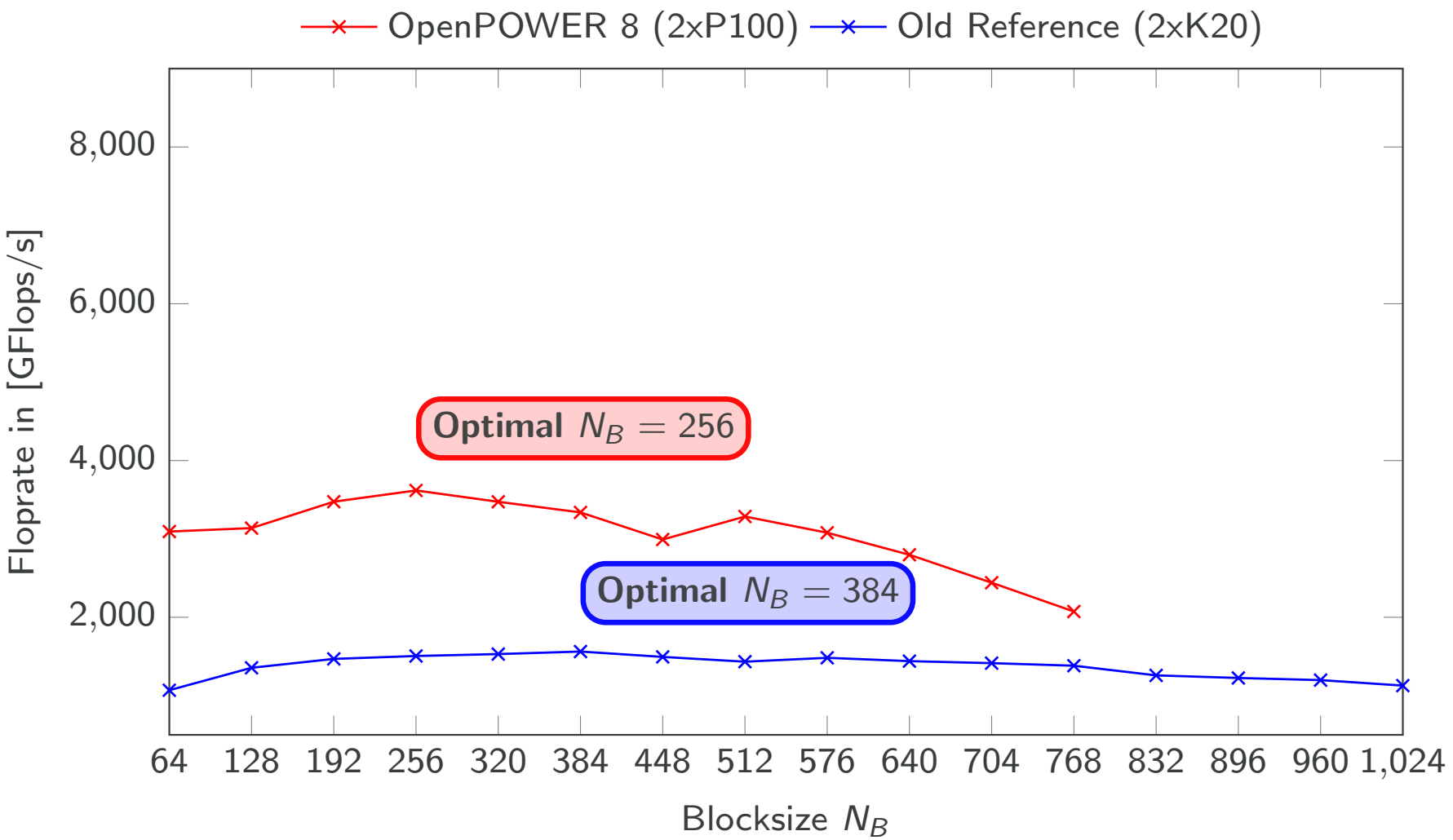
$n = 5,120$



Numerical Results

Optimal Blocksize

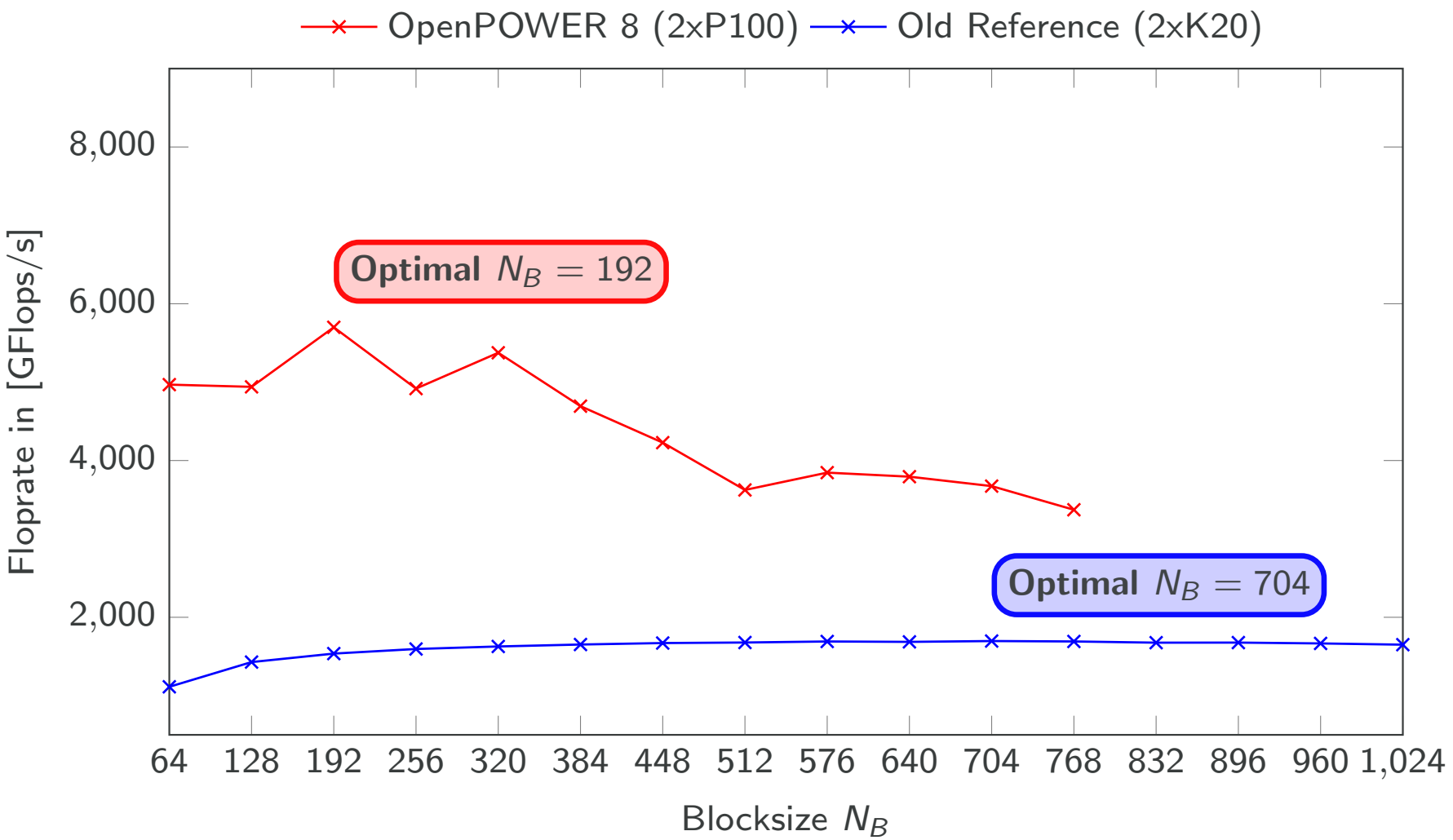
$n = 10,240$



Numerical Results

Optimal Blocksize

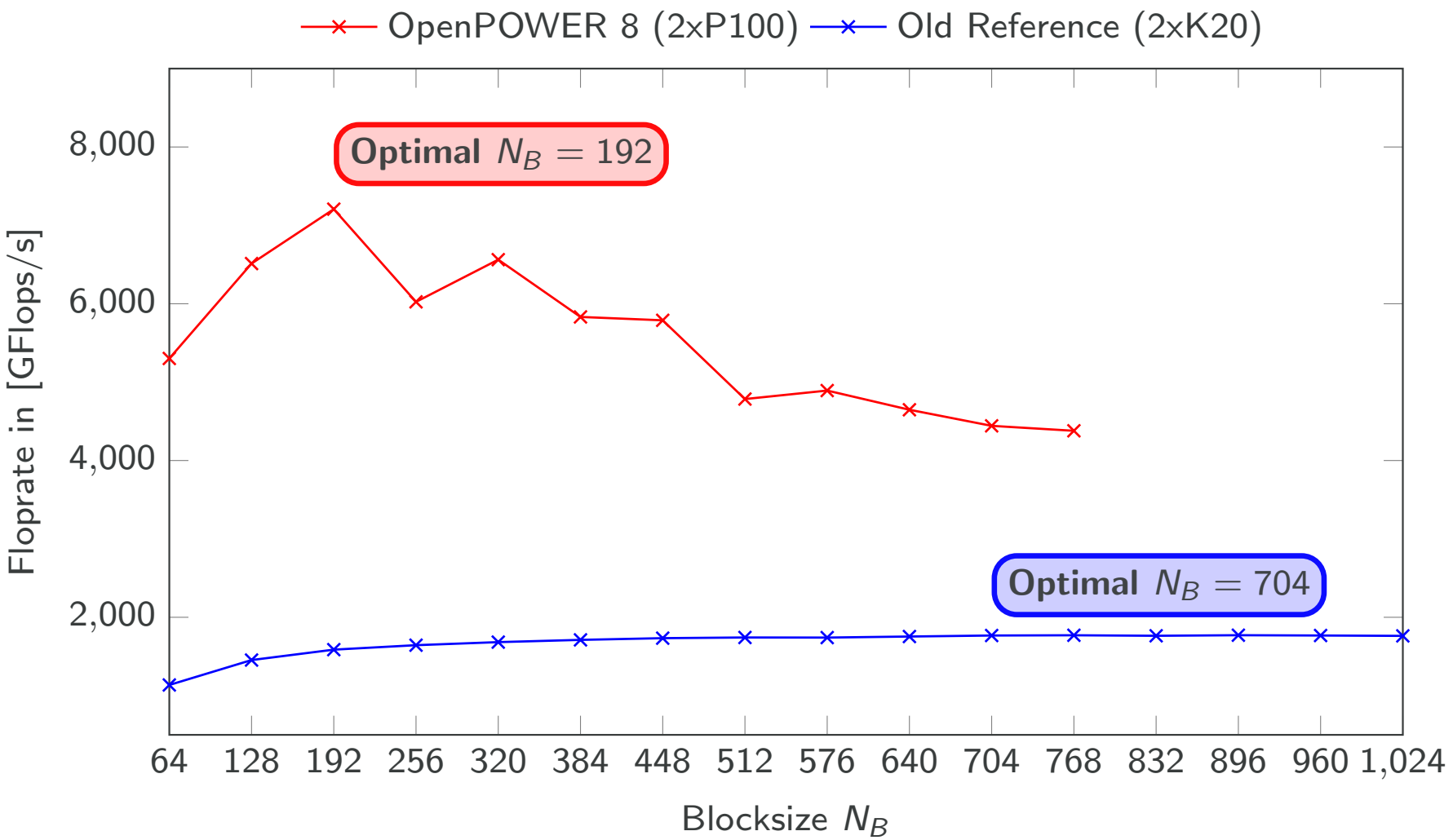
$n = 15,360$



Numerical Results

Optimal Blocksize

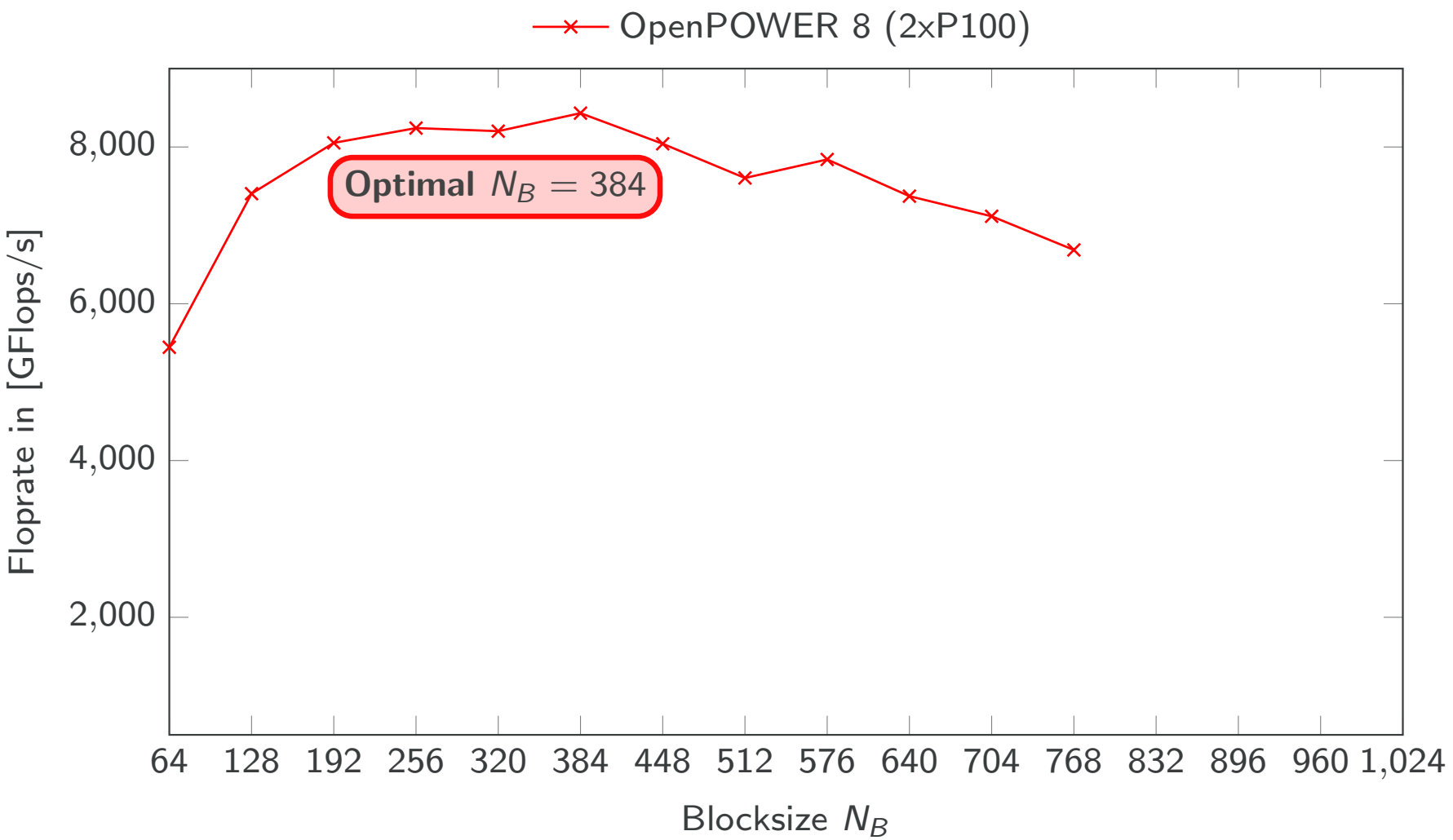
$n = 20,480$



Numerical Results

Optimal Blocksize

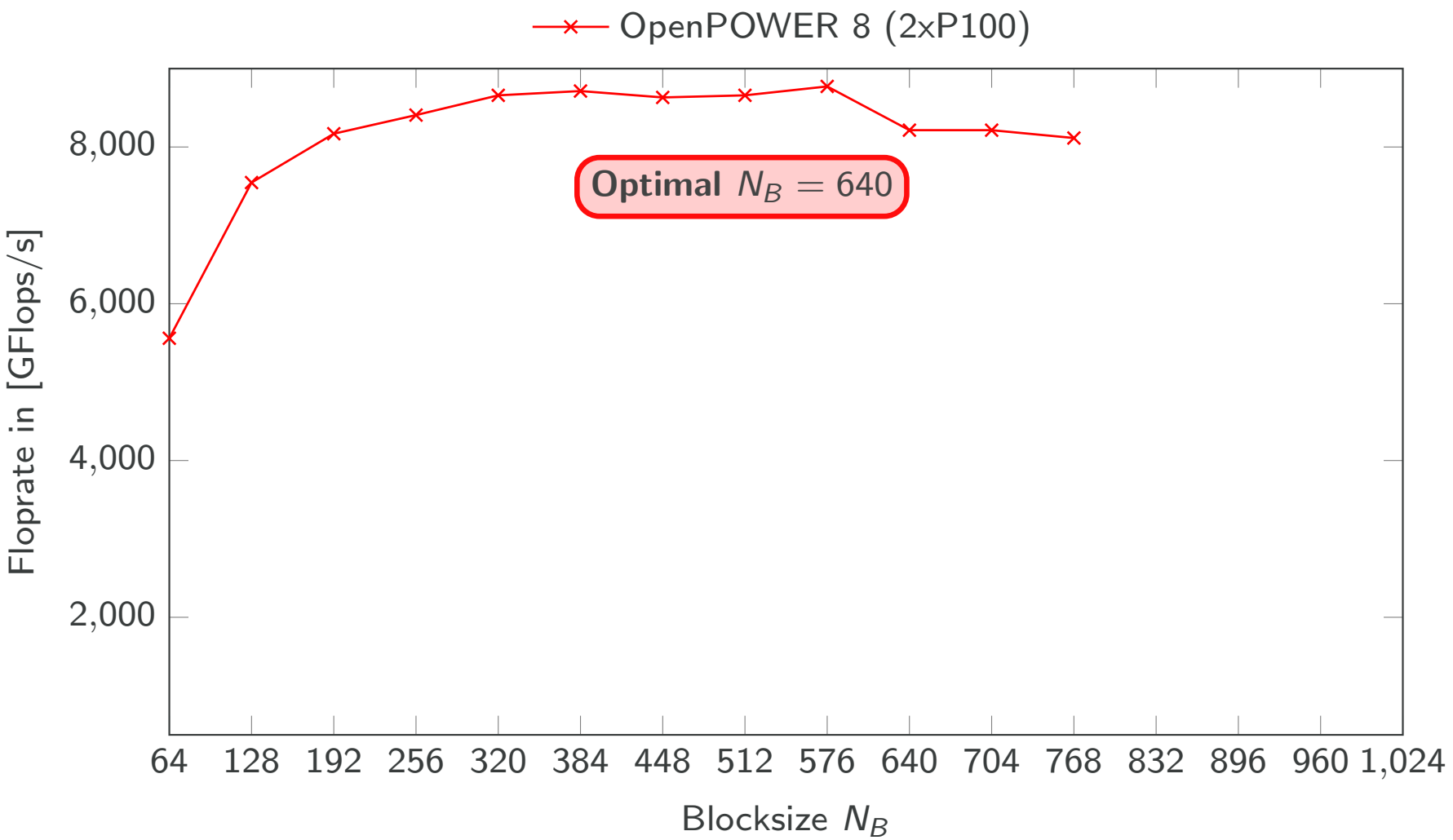
$n = 30,720$



Numerical Results

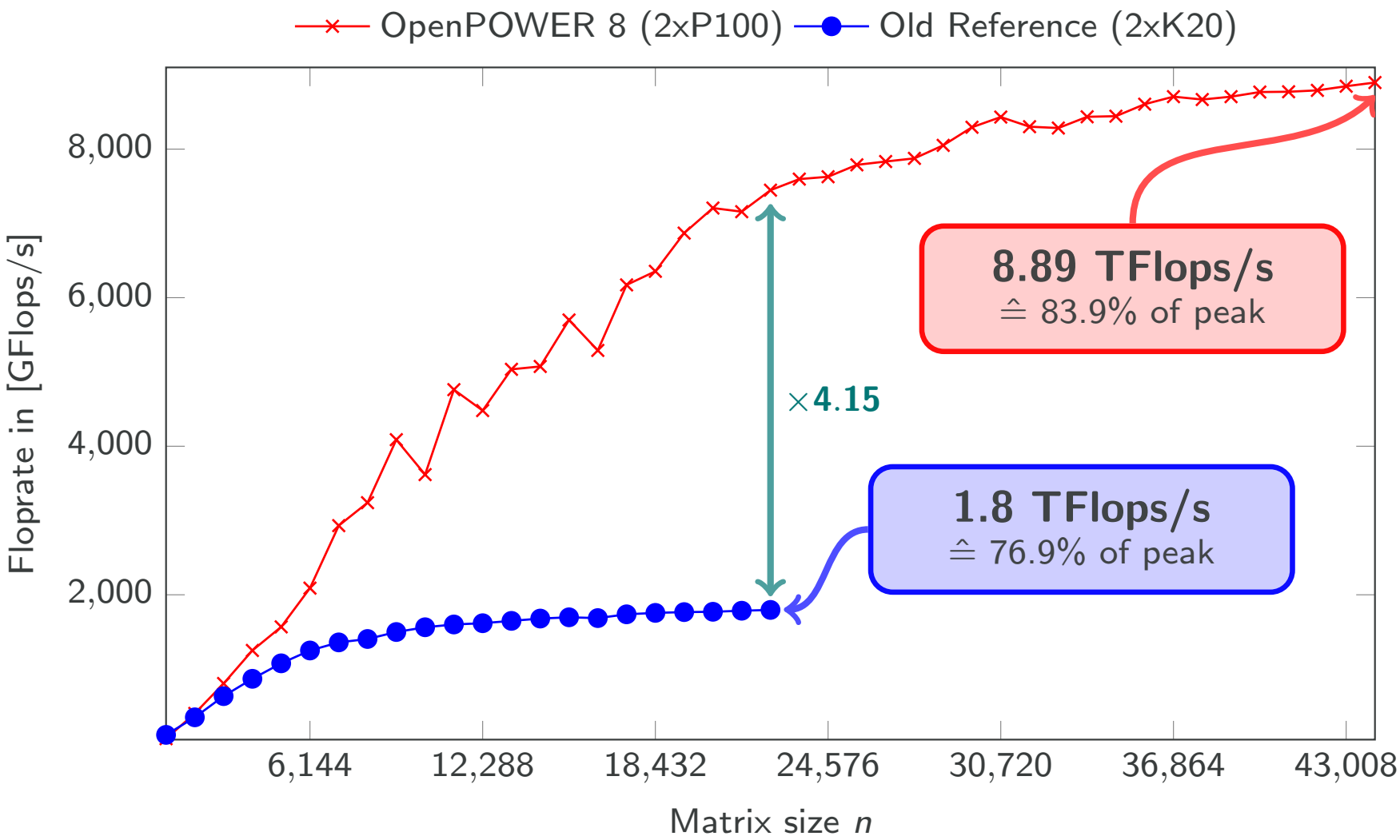
Optimal Blocksize

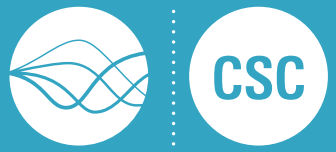
$n = 40,960$



Numerical Results

Performance with optimal blocksizes





Conclusions

Conclusions

- Speed up up to 5 between the K20 and the P100.
- Hybrid CPU-GPU algorithms are getting complicated due to the large performance differences.
- High bandwidth and smaller latencies yield smaller block sizes for optimal performance.
- "The Power System 822LC is an HPC Cluster in one server."

Conclusions

Conclusions

- Speed up up to 5 between the K20 and the P100.
- Hybrid CPU-GPU algorithms are getting complicated due to the large performance differences.
- High bandwidth and smaller latencies yield smaller block sizes for optimal performance.
- "The Power System 822LC is an HPC Cluster in one server."

Outlook and Future Work

- Implementation of an out-of-core solver
- Develop a fully integrated GPU accelerated matrix-sign function

Conclusions

Conclusions

- Speed up up to 5 between the K20 and the P100.
- Hybrid CPU-GPU algorithms are getting complicated due to the large performance differences.
- High bandwidth and smaller latencies yield smaller block sizes for optimal performance.
- "The Power System 822LC is an HPC Cluster in one server."

Thank you for your attention!

Outlook and Future Work

- Implementation of an out-of-core solver
- Develop a fully integrated GPU accelerated matrix-sign function