Kernverbesserungen von mitrax

- Eigener Datentyp für Zeilen und Spalten
- Elementrepräsentationsunabhängige Schnittstelle
  - Leicht erweiterbar für neue Arten der Datenrepräsentation (Adapter, Views, GPU ...)
- Intuitive und sichere Initialisierungssyntax
- „Zero-Overhead“ – Compiler kann bestmöglich optimieren
- Support von `constexpr`

Was macht eine Matrix aus?

- Anzahl Zeilen und Spalten; bekannt zur:
  - Compilezeit (mitrax, Eigen, Boost.uBLAS)
  - Laufzeit (mitrax, Eigen)
- Entsprechend rechteckig angeordnete Elemente gleichen Typs
  - Existieren im Speicher (mitrax, Eigen, Boost.uBLAS)
  - Zur Compilezeit bekannt (mitrax)
  - Erzeugung bei Zugriff (mitrax)

Bereitstellung der Elemente einer Matrix

- **constexpr**; Elemente sind zur Compilezeit bekannt und können direkt für Berechnungen verwendet werden. (mitrax)
- **Stack** (mitrax, Eigen)
- **Heap** (mitrax, Eigen, Boost.uBLAS)
- **View**; Das Matrix-Objekt besitzt die Daten nicht selbst, sondern ermöglicht nur den Zugriff. (mitrax)
- **Adapter**; Es wird ein anderer Datentyp (z. B. Eigen::Matrix) verwendet (mitrax)
- **Funktion**; Beim Zugriff wird der Wert eines Elementes entsprechend seiner Position in der Matrix und einer gegebenen Berechungsvorschrift berechnet. (mitrax)
- **GPU-Speicher** (teilweise mitrax, teilweise Eigen)
- **anderer Speicher**; Festplatte, FPGA ... (zukünftig mitrax)

Typisierte Zeilen & Spalten

- Eigener Typ für Zeilen und Spalten

```
struct col_t< 4 >
struct row_t< 7 >
```

- User-defined literals

```
4_C
7_R
```

- Typsicheres Rechnen mit Dimensionen

```
4_C + 3_C ✓
7_R + 2_C ✗
```

Intuitive & sichere Initialisierung

- Anforderungen:
  - Alle Elemente haben nach der Initialisierung einen definierten Wert
  - Direkte Initialisierung für kleine Matritzen
- Umsetzung:
  - Initialisierung mit Default-Value

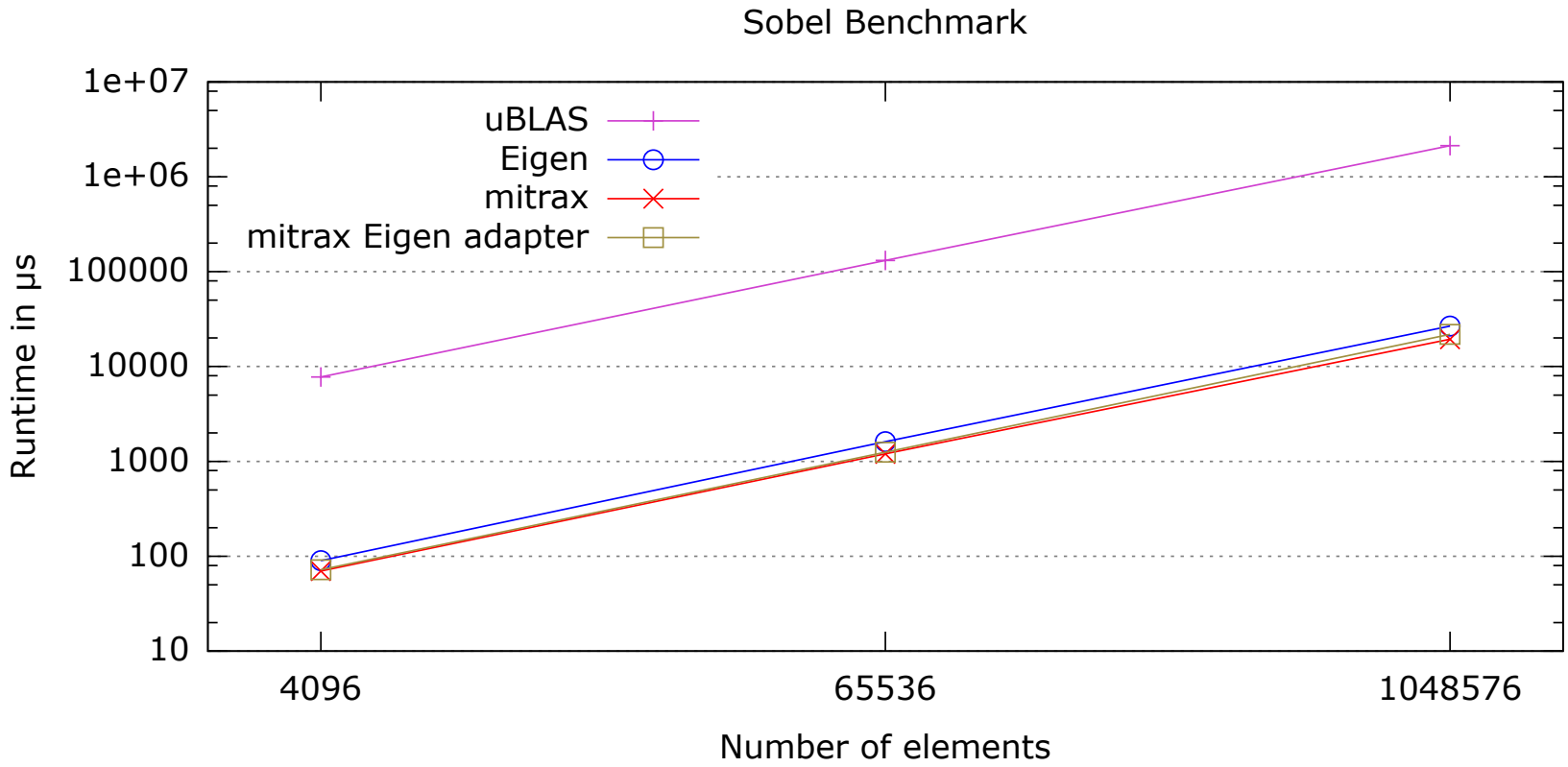
    ```
    auto matrix = make_matrix_v(3_C, 2_R, 3.f);
    ```
  - Direkte elementweise Initialisierung

    ```
    auto matrix = make_matrix(3_C, 2_R, {
        {1, 2, 3},
        {4, 5, 6}
    });
    ```

Anwendung: `constexpr` Matrix (1)

- Beim Sobel-Operator sind Dimensionen und Elemente der Faltungsmatrix zur Compilezeit bekannt
- Anmerkungen zum folgenden Benchmark:
  - Die Faltungsoperation ist für mitrax, Boost.uBLAS und Eigen identisch implementiert
  - Die Bildmatrix liegt immer auf dem Heap und hat Laufzeit-Dimensionen
  - Bei der Faltungsmatrix richtet sich dies nach der verwendeten Bibliothek:
    - mitrax: Compilezeit-Dimensionen, Daten sind constexpr
    - Eigen: Compilezeit-Dimensionen, Daten auf dem Stack
    - Boost.uBLAS: Laufzeit-Dimensionen, Daten auf dem Heap

Anwendung: `constexpr` Matrix (2)



Vektoren sind Matritzen

- Falls Zeilen oder Spalten zur Compilezeit den Wert 1 haben, werden durch die Schnittstelle zusätzliche Funktionen angeboten, ohne dass sich die Implementierung darum kümmern muss
  - Zugriffsoperator für Vektoren:

```
auto vector = make_matrix(1_C, 3_R, {{1}, {2}, {3}});
vector(0, 2) = 7; // normaler Matrix-Zugriff
vector[2] = 5;    // vereinfachter Vektor-Zugriff
```

  - Vereinfachte Erstellung:

```
auto vector = make_vector(3_R, {1, 2, 3});
```